



Урок 2

Асинхронность, пулы потоков, синхронизация

Оглавление

1	Использование пула потоков	2
2	Асинхронный вызов методов	10
3	Использование таймеров обратного вызова	31
4	Проблемы синхронизации.	35
5	Взаимоисключающий доступ. Класс Interlocked.	38
6	Критическая секция.	40
6.1	Класс Monitor.	42
6.2	Ключевое слово lock.	47
7	Мьютексы. Класс Mutex.	53
8	Семафоры. Класс Semaphore.	57
9	События.	59
9.1	Класс ManualResetEvent	60
9.2	Класс AutoResetEvent.	62
10	Практические примеры использования	62
11	Домашнее задание:	71

1 Использование пула потоков

И снова мы будем говорить о потоках. Я думаю, что по предыдущему уроку вы по достоинству оценили их полезность. Давайте еще раз отметим не только положительные, но и отрицательные стороны использования потоков. Если обратиться к истории (компьютерной), раньше операционные системы не поддерживали потоки. В однопоточных системах был лишь один поток, который попеременно выполнял все программы. Если в программе возникала ошибка, то она приводила к сбою всей системы. Так, небрежность при использовании бесконечного цикла в 16-разрядной версии Windows приводила к «зависанию» всей системы и заставляла принудительно перезагружаться. Поддержка многопоточности впервые была реализована в

Windows NT 3.1 с целью повышения отказоустойчивости системы. Здесь каждому процессу выделялся отдельный поток, и заикливание или зависание потока одного процесса не приводило к сбою потоков других процессов и всей системы.

Тем не менее, не смотря на свою полезность, потоки приводят к издержкам. Создание потока обходится недешево: объект ядра процесса должен быть размещен и инициализирован, для каждого потока выделяется 1 Мб адресного пространства для его стека пользовательского режима, также дополнительное пространство выделяется под стек режима ядра. Также при создании нового потока, Windows сразу же вызывает функцию в каждой DLL-библиотеке процесса, уведомляя все DLL-библиотеки о том, что поток был создан. При уничтожении потока каждая DLL-библиотека в процессе так же получает соответствующее уведомление об уничтожении потока. Также следует очистить объект ядра и стеки.

При выполнении программы проблем не меньше. Windows должна отслеживать объекты-потоки и время от времени решать, какой поток процессор будет обрабатывать следующим. А это дополнительный код, выполняющийся примерно раз в 20 мс. При определении очередного «счастливчика» необходима работа по остановке выполнения процессором кода одного потока и запуск на выполнение кода другого потока. То, что для нас звучит: «остановил-запустил» для системы означает выполнение следующих шагов:

1. Переход в режим ядра.

2. Сохранение регистров процессора в объекте ядра текущего процесса (а это до 1МБ памяти!!!).

3. Установка спин-блокировки и определение следующего потока для запуска. Затем освобождение спин-блокировки (при этом, если запускаемый поток относится к другому процессу затраты увеличиваются, т.к. происходит переключение виртуального адресного пространства).

4. Загрузка регистров процессора со значениями из объекта ядра потока, который будет выполняться следующим.

5. Выход из режима ядра.

Все это замедляет Windows и приложения по сравнению с однопоточной системой. Но это необходимая «жертва» во имя отказоустойчивости операционной системы.

Помимо повышения отказоустойчивости, потоки служат еще одной полезной цели — масштабируемости. В многопроцессорной системе Windows может планировать несколько потоков: по одному на каждый процессор. В таком случае имеет смысл создавать несколько потоков, чтобы эффективно задействовать возможности аппаратного обеспечения. Мы привыкли, что с каждым годом процессоры становились все мощнее, что не могло не сказаться на повышении производительности приложений. Но уже сейчас видно, что технология наращивания мощности на пределе. Поэтому

производители процессоров считают более эффективным создавать отдельные чипы, включающие несколько процессоров. Такие чипы повышают производительность многопоточных ОС и приложений за счет одновременного выполнения нескольких процессов. Современные производители процессорных чипов используют два типа технологий — Hyperthreading и многоядерную, — которые позволяют представить для Windows и приложений отдельный чип как два (и более) процессора.

В технологии Hyperthreading, физический процессор состоит из двух логических. У каждого из них есть своя архитектура (регистры процессора), но они пользуются общими рабочими ресурсами, например кэшем процессора. В идеале для процессоров, использующих Hyperthreading, можно рассчитывать на стопроцентное улучшение производительности, ведь теперь вместо одного процессора работают сразу два. Но, поскольку они используют общие рабочие ресурсы, ожидаемого резкого повышения производительности нет.

При использовании технологии нескольких ядер – существует несколько физических процессоров. У каждого — своя архитектура и рабочие ресурсы. Можно ожидать, что его рабочие характеристики будут лучше, чем у чипа Hyperthreading, и обеспечат стопроцентную производительность, потому что два процессора выполняют независимые задачи.

К этому месту главы вы должны были окончательно запутаться. Потоки – это хорошо, потому что система становится более отказоустойчива и быстра, но при этом потоки – это плохо, потому что расходуются ресурсы программного и аппаратного обеспечения. Ситуацию усугубляет еще и то, что производители процессоров идут по пути увеличения количества ядер на чипах.

Несмотря на всю грозность описанного выше, выход есть – это пул потоков. Пул потоков можно рассматривать как набор потоков, доступных

для использования приложением. У каждого процесса есть один пул потоков, который используется всеми доменами AppDomain этого процесса.

При инициализации CLR пул потоков пуст. Сам пул потоков обслуживает очередь запросов, поступающих от приложения. Когда приложению нужно выполнить асинхронную операцию, вызывается метод, который помещает запрос в очередь пула потоков. Концепция асинхронной операции заключается в том, что результат выполнения операции доступен не сразу же, а через некоторое время. Т.е. пока выполняется асинхронная операция, поток, который ее инициировал, может заниматься своими «делами». Таким образом, код пула потоков извлекает записи из очереди и передает их потоку из пула. Если пул пуст, создается новый поток. Как я уже рассказывал, создание потока снижает производительность. Но по завершении своей задачи поток из пула не уничтожается, а возвращается в пул и ожидает следующего запроса. Т.к. поток не уничтожается, производительность от этого не страдает. Когда приложение отправляет много запросов пулу потоков, пул пытается обработать все запросы с помощью одного потока. Но, если приложение создает очередь запросов, а поток из пула не успевает их обработать, создаются дополнительные потоки. В конце концов, все запросы приложения будут обрабатываться некоторым числом потоков, и пулу не придется создавать множество потоков.

Если же приложение перестанет создавать запросы пула потоков, в пуле может оказаться много незанятых потоков, которые будут попусту занимать ресурсы. Поэтому после некоторого бездействия поток пробуждается и самоуничтожается, чтобы освободить занимаемую память.

Пул потоков это компромисс в борьбе между экономией ресурсов и использованием многопоточности в многопроцессорных системах. На этом его преимущества не заканчиваются. Пул потоков действует по эвристическому алгоритму, приспособиваясь к текущей ситуации. Если приложение должно выполнить множество задач и есть доступные

процессоры, пул создает больше потоков. При уменьшении загруженности приложения потоки из пула самоуничтожаются.

В пуле различают два типа потоков: рабочие потоки и потоки ввода-вывода. Рабочие потоки используются, когда приложение запрашивает пул выполнить асинхронную вычислительную операцию. А потоки ввода-вывода служат для уведомления кода о завершении асинхронной операции ввода-вывода. Это значит, что для выполнения запросов, например обращения к файлу, сетевому серверу, базе данных, Web-службе или аппаратному устройству, используется модель APM (Asynchronous Programming Model).

До сих пор о пуле потоков упоминалось как о древнем божестве, но на самом деле описанная выше функциональность материализована в классе *System.Threading.ThreadPool*. Но прежде чем перейти к рассмотрению примера использования пула потоков, давайте разберемся с ограничением количества его потоков.

CLR позволяет разработчикам задавать максимальное число рабочих потоков и потоков ввода-вывода в пуле. Тем не менее, верхний предел числа потоков в пуле задавать не следует — это может привести к нехватке процессорного времени и даже к взаимной блокировке потоков.

На данный момент максимальное число рабочих потоков по умолчанию равно 25 на каждый процессор компьютера, а число потоков ввода-вывода — 1000. Полностью убирать эти ограничения нельзя, потому что в таком случае нарушится работа приложений, созданных для предыдущих версий CLR.

В классе *System.Threading.ThreadPool* есть несколько статических методов, которые служат для изменения количества потоков в пуле. Вот прототипы этих методов:

```
void GetMaxThreads(out int workerThreads, out int completionPortThreads);
```

```
bool SetMaxThreads(int workerThreads, int completionPortThreads);
```



```
void GetMinThreads(out int workerThreads, out int completionPortThreads);
```

```
bool SetMinThreads(int workerThreads, int completionPortThreads);
```

```
void GetAvailableThreads(out int workerThreads, out int completionPortThreads);
```

Метод *GetMaxThreads* позволяет узнать максимальное число потоков в пуле. Для изменения этого значения служит метод *SetMaxThreads*. Как правило, манипуляции с максимальным числом потоков в пуле обычно ухудшают, а не улучшают производительность приложения, поэтому оставляю работу с этими методами на вашу совесть или специфичность приложения.

Вместе с тем, в пул потоков CLR заложен механизм предотвращения слишком частого создания потоков — разрешается создавать не более одного потока в 500 мс. Если вас это не будет устраивать по причине замедления обработки запросов из очереди, то в приложение можно добавить вызов метода *SetMinThreads* и передать в него минимальное допустимое число потоков в пуле. Пул быстро создаст указанное число потоков, а если при появлении в очереди новых заданий все потоки будут заняты, он продолжит создавать потоки со скоростью не более одного потока в 500 мс.

Как же все-таки использовать пул потоков и асинхронную модель программирования? Рекомендация следующая: в общем случае при выполнении вычислительных операций не должны происходить никакие синхронные операции ввода-вывода, потому что они приостанавливают вызывающий поток на время работы аппаратных. Нужно всегда пытаться сделать так, чтобы потоки были активны. Поток, который приостановлен – не выполняет никакой полезной работы, но попусту использует ресурсы системы — именно этого следует избегать при создании высокопроизводительного приложения.



Чтобы добавить в очередь пула потоков асинхронную вычислительную операцию, обычно вызывают один из следующих методов класса

ThreadPool:

```
static bool QueueUserWorkItem(WaitCallback callBack);
```

```
static bool QueueUserWorkItem(WaitCallback callBack, object state);
```

```
static bool UnsafeQueueUserWorkItem(WaitCallback callBack, object state);
```

Эти методы ставят «рабочий элемент» в очередь пула потоков и сразу возвращают управление приложению. Под рабочим элементом подразумевают указанный в параметре *callBack* метод, который вызывается потоком из пула. Этому методу можно передать один параметр, указанный в параметре *state*. Версия метода *QueueUserWorkItem* без параметра *state* передает *null* методу обратного вызова. В конце концов, один из потоков пула обработает рабочий элемент, что приведет к вызову указанного метода. Создаваемый метод обратного вызова должен соответствовать типу-делегату *System.Threading.WaitCallback*, который определяется так:

```
delegate void WaitCallback(object state);
```

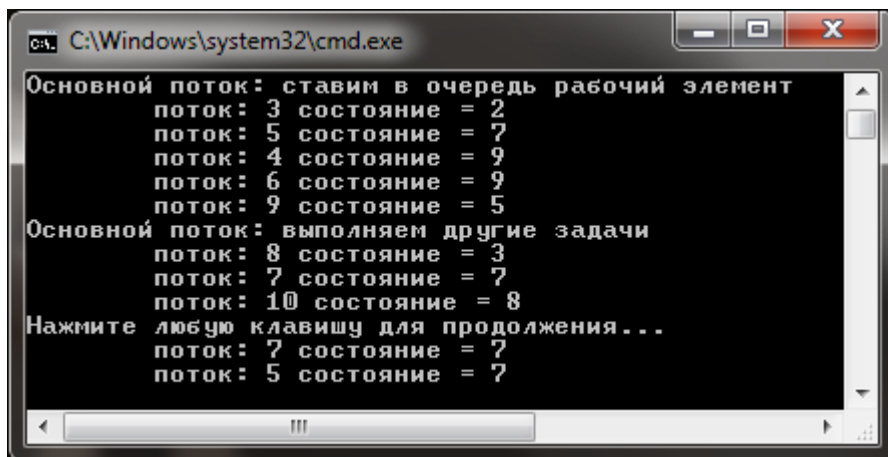
Анализируем пример работы пула потоков:


```
using System;
using System.Threading;

namespace ConsoleApplication1
{
    class PoolUsingClass
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Основной поток: ставим в очередь рабочий элемент");
            Random r = new Random();
            for (int i = 0; i < 10; ++i)
                ThreadPool.QueueUserWorkItem(WorkingElementMethod, r.Next(10));
            Console.WriteLine("Основной поток: выполняем другие задачи");
            Thread.Sleep(1000);
            Console.WriteLine("Нажмите любую клавишу для продолжения...");
            Console.ReadLine();
        }

        private static void WorkingElementMethod(object state)
        {
            Console.WriteLine("\tpоток: {0} состояние = {1}",
                Thread.CurrentThread.ManagedThreadId, state);
            Thread.Sleep(1000);
        }
    }
}
```

В результате выполнения этого кода получим следующие результаты:



```
C:\Windows\system32\cmd.exe
Основной поток: ставим в очередь рабочий элемент
    поток: 3 состояние = 2
    поток: 5 состояние = 7
    поток: 4 состояние = 9
    поток: 6 состояние = 9
    поток: 9 состояние = 5
Основной поток: выполняем другие задачи
    поток: 8 состояние = 3
    поток: 7 состояние = 7
    поток: 10 состояние = 8
Нажмите любую клавишу для продолжения...
    поток: 7 состояние = 7
    поток: 5 состояние = 7
```

На самом деле результаты могут быть и другими: в зависимости от количества ядер процессора и других факторов. Это объясняется тем, что методы выполняются асинхронно. Из примера видно, что использовать поток

из пула потоков не сложнее, чем создавать отдельный поток. Тогда возникает вопрос: почему бы не отказаться от использования обычных потоков в пользу «пуловых»? Здесь рекомендация следующая: применяйте пул потоков для выполнения асинхронных операций везде, где это возможно, за исключением тех случаев, когда нужно выполнить код, требующий особого режима потока, нетипичного для потоков из пула. В частности, выделенный поток создается для выполнения задачи с определенным приоритетом. Ведь все потоки из пула выполняются со стандартным приоритетом, который не следует изменять.

Также имеет смысл создать выделенный поток, если нужно, чтобы поток работал в активном (foreground) режиме. В отличие от этого, все потоки из пула выполняются в фоновом режиме и никак по-другому. В этом случае приложение завершится лишь после того, как поток выполнит свою задачу. Выделенный поток также пригодится для вычислительной задачи, требующей очень много времени. Это освободит логику пула потоков от необходимости решать вопрос о создании дополнительных потоков. И, наконец, выделенный поток нужен, когда может потребоваться завершить его досрочно, вызвав метод *Abort* объекта *Thread*.

2 Асинхронный вызов методов

В предыдущей главе я упоминал об асинхронной модели программирования. Давайте более детально разберем, что это за «зверь». Под моделью асинхронного программирования понимают модель создания высокопроизводительных приложений, использующих асинхронные операции вместе с пулом потоков. Простота и эффективность этой модели не могла не затронуть библиотеку основных классов FCL (FrameWork Class



Library). Здесь эту модель использует большое количество классов: все классы производные от:

- *System.IO.Stream*
- *System.Net.Dns*
- *System.Net.Sockets.Socket*
- *System.Net.WebRequest*
- *System.IO.Ports.SerialPort*
- *System.Data.SqlClient.SqlCommand* и прочие.

Так же есть возможность, создавая свои типы, – «приобщать» их к этой модели. Рассмотрим все по порядку.

Сначала рассмотрим стандартные асинхронные методы стандартных классов.

Чтобы синхронно прочесть байты из объекта *FileStream*, нужно вызвать метод *Read*. Вот его прототип:

```
public int Read(byte[] array, int offset, int count)
```

Метод *Read* принимает ссылку на массив *byte[]*, в который будут записываться байты из файла. Параметр *count* задает максимальное число байт, которые нужно прочесть. Байты помещаются в массив *array* под номерами от *offset* до *(offset + count - 1)*.

Метод *Read* возвращает количество байтов, фактически прочитанных из файла. При вызове этого метода чтение выполняется синхронно. Это значит, что метод возвращает управление приложению лишь после того, как все запрошенные байты считаны в байтовый массив. Синхронная операция ввода-вывода очень неэффективна, потому как время ее выполнения непредсказуемо, а вызывающий поток приостанавливается до ее завершения, поэтому не может выполнять другие задачи и впустую расходует ресурсы.



Если Windows сохранила содержимое файла в кэше, этот метод почти сразу вернет управление приложению. Но если нужных данных в кэше не окажется, то Windows придется обращаться к жесткому диску, чтобы загрузить с него содержимое файла.

Может случиться так, что операционной системе придется обращаться по сети к серверу, чтобы тот обратился к своему жесткому диску (или кэшу) и вернул нужные данные.

Для исправления этой ситуации нужно прочесть содержимое файла асинхронно. Для этого создаем объект *System.IO.FileStream*, используя перегрузку конструктора, в которой присутствует параметр *System.IO.FileOptions*. В этом параметре нужно указать флаг *FileOptions.Asynchronous*, который сообщает объекту *FileStream*, что нужно выполнить асинхронную операцию чтения и записи в файле.

Теперь для асинхронного чтения необходимого количества байтов из файла необходимо вызвать метод *BeginRead* объекта *FileStream*:

```
IAsyncResult BeginRead(byte[] array, int offset, int numBytes, AsyncCallback  
                        userCallback, object stateObject)
```

Заметьте – первые три параметра у методов *BeginRead* и *Read* совпадают. Но у *BeginRead* есть еще два параметра — *userCallback* и *stateObject*.

Вызов метода *BeginRead* дает Windows команду прочесть байты из файла в байтовый массив. Поскольку это операция ввода-вывода, метод *BeginRead* ставит этот запрос в очередь драйвера соответствующего аппаратного устройства. Вся работа ложится на аппаратное устройство, и потокам больше ничего не нужно делать — даже ждать результатов операции. Т.е. вызывающий поток не приостанавливается, а может дальше выполнять свою работу.



Метод *BeginRead* возвращает ссылку на объект, тип которого реализует интерфейс *System.IAsyncResult*. При вызове метода *BeginRead* он создает объект, который уникально идентифицирует запрос на ввод-вывод, ставит запрос в очередь драйвера устройства Windows, а затем возвращает объект *IAsyncResult*. Этот объект похож на расписку в получении. Когда метод *BeginRead* возвращает управление приложению, операция ввода-вывода только была поставлена в очередь, но еще не завершена. Поэтому не пытайтесь использовать байты из байтового массива — ведь в нем еще может не быть запрошенных данных.

На самом деле, массив уже может содержать запрошенные данные, потому что операция ввода-вывода выполняется асинхронно и к моменту завершения работы метода *BeginRead* чтение нужных данных может быть завершено. Так же возможна ситуация, что данные поступят с сервера через несколько секунд. Но может случиться и так, что произойдет сбой подключения к серверу, и данные не поступят никогда. Поскольку все эти ситуации возможны, необходим способ определить, что и когда произошло на самом деле. Существуют три способа это выяснить.

Первый способ – это ожидание завершения. Не имеет значения, какой из стандартных классов захотел использовать асинхронную операцию, но в любом случае она инициируется вызовом метода, название которого начинается с *Begin* (*BeginXXX*). Такие методы ставят операцию в очередь и возвращают объект *IAsyncResult*, указывающий на выполняемую операцию. Для получения результата операции нужно просто вызвать соответствующий метод *EndXXX*, передав ему объект *IAsyncResult*. При вызове метода *EndXXX* CLR получает команду вернуть результат асинхронной операции, на которую указывает объект *IAsyncResult* (полученный в результате вызова метода *BeginXXX*).



Таким образом, если на момент вызова метода *EndXXX* асинхронная операция уже завершилась, он сразу вернет ее результат. В противном случае *EndXXX* приостановит вызывающий поток до завершения операции, а затем вернет результат.

Пробуем применить этот механизм при асинхронном считывании данных из файла.

В классе *FileStream* есть метод *EndRead*:

```
int EndRead(IAsyncResult asyncResult);
```

Как и было отмечено выше – у него всего один параметр — *IAsyncResult*. Тип возврата совпадает с синхронным методом *Read*, и также означает число фактически считанных байтов.

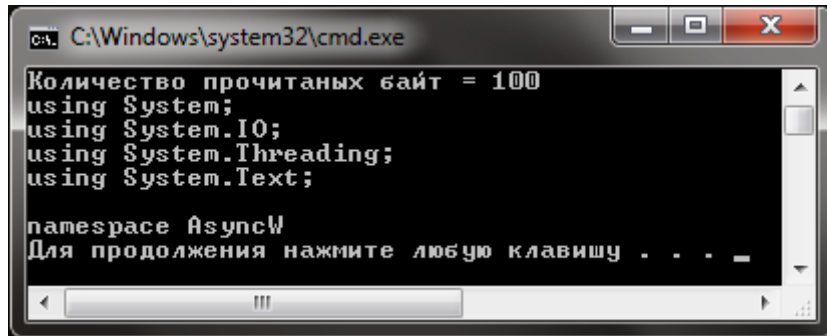
Рассмотрим этот способ на примере:

```
class AsyncWaitClass
{
    static void Main(string[] args)
    {
        FileStream fs = new FileStream(@"../../Program.cs", FileMode.Open,
                                       FileAccess.Read, FileShare.Read, 1024,
                                       FileOptions.Asynchronous);

        byte[] data = new byte[100];
        // Начало асинхронной операции чтения из файла FileStream.
        IAsyncResult ar = fs.BeginRead(data, 0, data.Length, null, null);
        // Здесь выполняется другой код...
        // Приостановка этого потока до завершения асинхронной операции
        // и получения ее результата.
        Int bytesRead = fs.EndRead(ar);
        // Других операций нет. Закрытие файла.
        fs.Close();
        // Теперь можно обратиться к байтовому массиву и вывести результат операции.
        Console.WriteLine("Количество прочитанных байт = {0}", bytesRead);

        Console.WriteLine(Encoding.UTF8.GetString(data));
    }
}
```

В результате выполнения получим:



```
C:\Windows\system32\cmd.exe
Количество прочитанных байт = 100
using System;
using System.IO;
using System.Threading;
using System.Text;

namespace AsyncW
Для продолжения нажмите любую клавишу . . . _
```

На самом деле этот пример не совсем образцовый. Поскольку между вызовами методов *BeginRead* и *EndRead* нет никакого кода, то в асинхронном чтении смысла нет, т.к. вызывающий поток приостанавливается для ожидания. Тут можно было смело использовать синхронный метод *Read*. Попробуем несколько улучшить это приложение и добавим одновременное чтение их нескольких файлов.



```
using System;
using System.IO;
using System.Threading;
using System.Text;

namespace AsyncWait
{
    class AsyncWaitClass
    {
        static void Main(string[] args)
        {
            string[] files = {"../..//Program.cs",
                              "../..//AsyncWait.csproj",
                              "../..//Properties/AssemblyInfo.cs"};
            AsyncReader[] asrArr = new AsyncReader[3];
            for(int i = 0; i < asrArr.Length; ++i)
                asrArr[i] = new AsyncReader(new FileStream(files[i], FileMode.Open,
                                                            FileAccess.Read,
                                                            FileOptions.Asynchronous), 100);

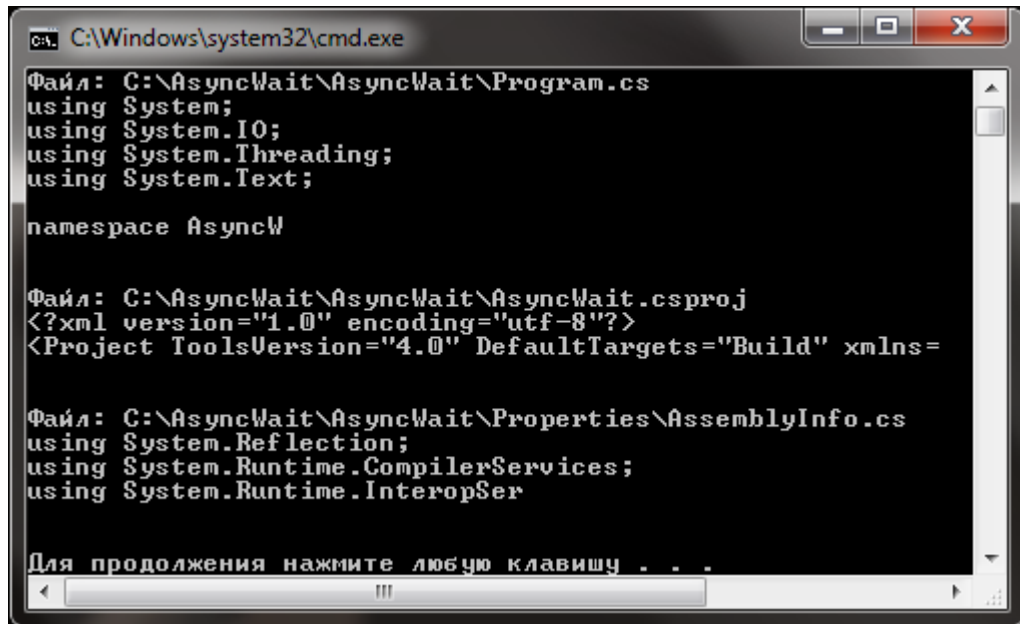
            foreach (AsyncReader asr in asrArr)
                Console.WriteLine(asr.EndRead());
        }
    }

    class AsyncReader
    {
        FileStream stream;
        byte[] data;
        IAsyncResult asRes;

        public AsyncReader(FileStream s, int size)
        {
            stream = s;
            data = new byte[size];
            asRes = s.BeginRead(data, 0, size, null, null);
        }

        public string EndRead()
        {
            int countByte = stream.EndRead(asRes);
            stream.Close();
            Array.Resize(ref data, countByte);
            return string.Format("Файл: {0}\n{1}\n\n", stream.Name,
                                Encoding.UTF8.GetString(data));
        }
    }
}
```


В результате выполнения приложения получим следующие результаты:



```
C:\Windows\system32\cmd.exe

Файл: C:\AsyncWait\AsyncWait\Program.cs
using System;
using System.IO;
using System.Threading;
using System.Text;

namespace AsyncW

Файл: C:\AsyncWait\AsyncWait\AsyncWait.csproj
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns=

Файл: C:\AsyncWait\AsyncWait\Properties\AssemblyInfo.cs
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices

Для продолжения нажмите любую клавишу . . .
```

Теперь этот код выполняет все операции чтения одновременно, но все-таки он не стал полностью образцовым. Дело в том, что после постановки в очередь всех запросов на чтение выполнение переходит ко второму циклу, в конце которого последовательно вызывается метод *EndRead* для каждого потока в порядке поступления запросов на чтение. Но поскольку время чтения данных в разных потоках может отличаться, то это неэффективно. Возможна ситуация, когда данные от второго потока поступят раньше данных от первого потока. В таком случае хорошо бы сначала обработать данные от второго потока, а потом, по мере их поступления, от первого.

Второй способ определения завершения асинхронной операции основывается на опросе состояния асинхронного запроса. Он несколько поможет нам исправить недочеты первого способа, но также добавит новые проблемы.



Итого, у объекта *IAAsyncResult* есть свойство *IsCompleted*, которое имеет следующий прототип:

```
bool IsCompleted { get; }
```

Вызывающий поток, с помощью данного свойства, периодически осведомляется у CLR, завершен ли асинхронный запрос. При этом время работы потока расходуется впустую. Именно по этому данный подход не является «эстетическим». Тем не менее, если мы можем пожертвовать производительностью во имя простоты кода, то этот подход имеет место быть. Рассмотрим подробнее код с использованием способа регулярного опроса:

```
namespace AsyncRequest
{
    class AsyncRequestClass
    {
        static void Main(string[] args)
        {
            FileStream fs = new FileStream(@"../../Program.cs", FileMode.Open,
                FileAccess.Read, FileShare.Read, 1024,
                FileOptions.Asynchronous);

            Byte[] data = new Byte[100];

            IAsyncResult ar = fs.BeginRead(data, 0, data.Length, null, null);

            while (!ar.IsCompleted)
            {
                Console.WriteLine("Операция не завершена, ожидайте...");
                Thread.Sleep(10);
            }

            Int32 bytesRead = fs.EndRead(ar);

            fs.Close();

            Console.WriteLine("Количество считанных байт = {0}", bytesRead);
            Console.WriteLine(Encoding.UTF8.GetString(data).Remove(0, 1));
        }
    }
}
```



Результат выполнения этого кода идентичен результату выполнения кода который читал содержимое файла Program.cs.

Цикл, который стоит после вызова метода *BeginRead*, периодически опрашивает свойство *IsCompleted* объекта *IAAsyncResult*, и возвращает значение *false*, если асинхронная операция еще не завершилась, и *true* в противном случае.

Если возвращается ложное значение, то управление переходит в блок *if*, где вызывается метод *sleep*, чтобы приостановить поток, иначе он будет прокручивать этот цикл, впустую расходуя процессорное время. Если же асинхронная операция завершится до того, как цикл впервые запросит свойство *IsCompleted*, оно вернет значение *true* и цикл не будет выполняться вообще.

Рано или поздно асинхронная операция будет выполнена, свойство *IsCompleted* вернет значение *true*, и цикл завершится. После этого вызывается метод *EndRead* для получения результатов.

Вот другой пример использования метода регулярного опроса, в котором запрашивается свойство *AsyncWaitHandle* объекта *IAAsyncResult*.

Свойство *AsyncWaitHandle* объекта *IAAsyncResult* возвращает ссылку на объект, производный от *WaitHandle*, — обычно это *System.Threading.ManualResetEvent*.

Цикл в этой программе отличается от цикла из предыдущего примера тем, что вызов метода *sleep* для потока не нужен, потому что методу *WaitOne* в качестве параметра передано значение времени приостановки потока — 10 мс.



```
using System;
using System.IO;
using System.Threading;
using System.Text;

namespace AsyncRequest
{
    class AsyncRequestClass
    {
        FileStream fs = new FileStream(@"../../Program.cs", FileMode.Open,
                                       FileAccess.Read, FileShare.Read, 1024,
                                       FileOptions.Asynchronous);

        Byte[] data = new Byte[100];

        IAsyncResult ar = fs.BeginRead(data, 0, data.Length, null, null);

        while (!ar.AsyncWaitHandle.WaitOne(10, false))
        {
            Console.WriteLine("Операция не завершена, ожидайте...");
        }

        Int32 bytesRead = fs.EndRead(ar);

        fs.Close();

        Console.WriteLine("Количество считанных байт = {0}", bytesRead);
        Console.WriteLine(Encoding.UTF8.GetString(data).Remove(0, 1));
    }
}
```

Результат выполнения этого кода опять идентичен предыдущему, т.к. выполняются одни и те же действия только разными способами.

Т.к. способ с опросом через свойство *IsCompleted* впустую тратит процессорное время, а способ с применением свойства *AsyncWaitHandle* работает медленно из-за использования объекта производного от *WaitHandle* (об этом будет рассказано далее), то ни один из описанных выше подходов до сих пор не претендует на звание самого лучшего. А по сему рассмотрим последний метод, который является наилучшим из показанных.

Данный подход использует метод обратного вызова. Из всех способов получения результата в асинхронной модели программирования этот способ

лучше всего подходит для создания приложения с высокими требованиями к производительности и масштабируемости.

Причина этого состоит в том, что при использовании этого подхода поток никогда не переводится в режим ожидания (в отличие от метода ожидания завершения) и не происходит напрасного расходования ресурсов процессора, когда программа периодически проверяет окончание асинхронной операции (как в методе последовательного опроса).

Принцип работы этого метода следующий: сначала запрос на асинхронный ввод-вывод ставится в очередь, а затем вызывающий поток выполняет любую другую работу. Когда запрос ввода-вывода выполнится, Windows поставит «рабочий элемент» в очередь пула потоков CLR. В конечном итоге поток из пула выберет из очереди «рабочий элемент» и вызовет определенный разработчиком метод — так приложение узнает о завершении асинхронного ввода-вывода. Таким образом, для получения результатов асинхронной операции в методе обратного вызова сначала вызывается метод *BeginXXX*, после чего метод обратного вызова может продолжить обработку результата. Когда метод возвращает управление программе, поток возвращается в пул и готов к обслуживанию следующего «рабочего элемента» в очереди (если таковой имеется в очереди запросов).

Теперь посмотрим практический пример. Прототип метода *BeginRead* объекта *FileStream* включает два последних параметра, значения которым мы до сих пор устанавливали в *null*. Это параметры *userCallback* и *stateObject*. Эти параметры представляют собой тип-делегат, определяемый так:

```
delegate void AsyncCallback(IAsyncResult ar);
```

Этот делегат определяет сигнатуру метода обратного вызова, которую нужно реализовать. В качестве параметра *stateObject* метода *BeginXXX*



можно передать любое значение. Этот параметр просто позволяет передать некие данные от метода, ставящего операцию в очередь, методу обратного вызова, обрабатывающему завершение этой операции. Метод обратного вызова получает ссылку на объект *IAsyncResult*, он также может получить ссылку на статический объект, запросив свойство *AsyncState* объекта *IAsyncResult*. Следующий пример получение результата асинхронной операции с использованием способа обратного вызова метода.

```
namespace AsyncReadCallBack
{
    class AsyncReadCallBackClass
    {
        private static Byte[] data = new Byte[100];

        static void Main(string[] args)
        {
            Console.WriteLine("Основной поток ID = {0}",
                Thread.CurrentThread.ManagedThreadId);

            FileStream fs = new FileStream(@"../../Program.cs", FileMode.Open,
                FileAccess.Read, FileShare.Read, 1024,
                FileOptions.Asynchronous);

            fs.BeginRead(data, 0, data.Length, ReadIsComplete, fs);

            Console.ReadLine();
        }

        private static void ReadIsComplete(IAsyncResult ar)
        {
            Console.WriteLine("Чтение в потоке {0} закончено",
                Thread.CurrentThread.ManagedThreadId);

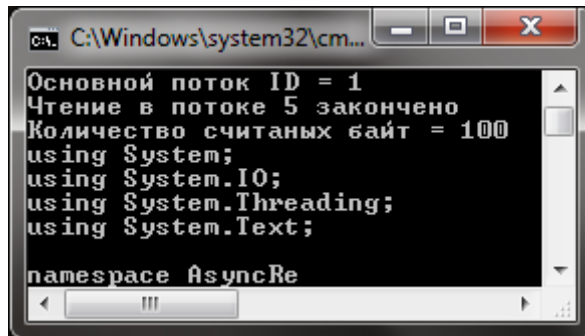
            FileStream fs = (FileStream)ar.AsyncState;

            Int32 bytesRead = fs.EndRead(ar);

            fs.Close();

            Console.WriteLine("Количество считанных байт = {0}", bytesRead);
            Console.WriteLine(Encoding.UTF8.GetString(data).Remove(0, 1));
        }
    }
}
```

Результат выполнения следующий:



```
C:\Windows\system32\cmd...
Основной поток ID = 1
Чтение в потоке 5 закончено
Количество считанных байт = 100
using System;
using System.IO;
using System.Threading;
using System.Text;
namespace AsyncRe
```

Прежде всего, обратите внимание, что метод *Main* выполнялся основным потоком с идентификатором 1, а *ReadIsDone* — потоком из пула с идентификатором 5 (данные могут отличаться).

Таким образом, в выполнении этой программы участвовали два различных потока. Следующий положительный момент – это то, что *IAsyncResult*, возвращаемый методом *BeginRead*, не сохраняется в переменной в методе *Main*. Имя метода обратного вызова передается в качестве четвертого параметра методу *BeginRead*. И наконец, *fs* передается последним параметром методу *BeginRead*. Метод обратного вызова получает ссылку на *FileStream*, запрашивая свойство *AsyncState* переданного объекта *IAsyncResult*.

Данная версия программы работает идеально, но тем не менее нет предела совершенству и как всегда можно добавить последний штрих. Поле *data* было объявлено статическим, чтобы к нему можно было обращаться из методов *Main* и *ReadIsDone*. Это есть не совсем хорошо, потому как когда-нибудь нужно будет выполнить несколько запросов ввода-вывода — тогда лучше всего динамически создавать *Byte[]* для каждого запроса. Решить эту задачу можно двумя способами:

Определяем отдельный класс, содержащий поля *Byte[]* и *FileStream*, создаем экземпляр этого класса, инициализируем поля и передаем эту ссылку как последний параметр метода *BeginRead*. Тогда метод обратного



вызова сможет обращаться к обоим источникам данных и использовать их внутри метода.

Более изящный вариант – с использованием анонимных методов.

Вот как эта программа выглядит при использовании анонимных методов:

```
using System;
using System.IO;
using System.Threading;
using System.Text;

namespace AsyncReadCallBack
{
    class AsyncReadCallBackClass
    {
        private static Byte[] staticData = new Byte[100];

        static void Main(string[] args)
        {
            Byte[] data = new Byte[100];

            Console.WriteLine("Основной поток ID = {0}",
                Thread.CurrentThread.ManagedThreadId);

            FileStream fs = new FileStream(@"../../Program.cs", FileMode.Open,
                FileAccess.Read, FileShare.Read, 1024,
                FileOptions.Asynchronous);

            fs.BeginRead(data, 0, data.Length,
                delegate(IAsyncResult ar)
                {
                    Console.WriteLine("Чтение в потоке {0} закончено",
                        Thread.CurrentThread.ManagedThreadId);

                    Int32 bytesRead = fs.EndRead(ar);
                    fs.Close();

                    Console.WriteLine("Количество считанных байт = {0}",
                        bytesRead);

                    Console.WriteLine(Encoding.UTF8.GetString(data));
                    Console.ReadLine();
                }, null);
            Console.ReadLine();
        }
    }
}
```




Результат выполнения кода такой же, как и без применения анонимных методов.

Отличие данного варианта программы в том, что в качестве последнего параметра метода *BeginRead* передается *null*. Это возможно из-за того, что анонимный метод может обращаться к любой локальной, определенной в методе *Main*.

Ну и на остаток темы использования стандартных асинхронных методов – улучшим, за счет методов обратного вызова и анонимных методов, программу, которая асинхронно считывала данные из нескольких файлов. Старый вариант был недостаточно эффективен, поскольку он обрабатывал данные в порядке постановки в очередь, а не в порядке их поступления.

```
using System;
using System.IO;
using System.Threading;
using System.Text;

namespace AsyncReadCallBack
{
    class AsyncReadCallBackClass
    {
        private static Byte[] staticData = new Byte[100];

        static void Main(string[] args)
        {
            string[] files = {"../Program.cs",
                              "../AsyncReadCallBack.csproj",
                              "../Properties/AssemblyInfo.cs"};

            for (int i = 0; i < files.Length; ++i)
                new AsyncCallBackReader(new FileStream(files[i],
                                                         FileMode.Open, FileAccess.Read, FileShare.Read, 1024,
                                                         FileOptions.Asynchronous), 100,
                                         delegate(Byte[] data)
                                         {
                                             Console.WriteLine("Количество прочитанных байт = {0}",
                                                                     data.Length);

                                             Console.WriteLine(Encoding.UTF8.GetString(data) +
                                                                     "\n\n");
                                         });
            Console.ReadLine();
        }
    }
}
```

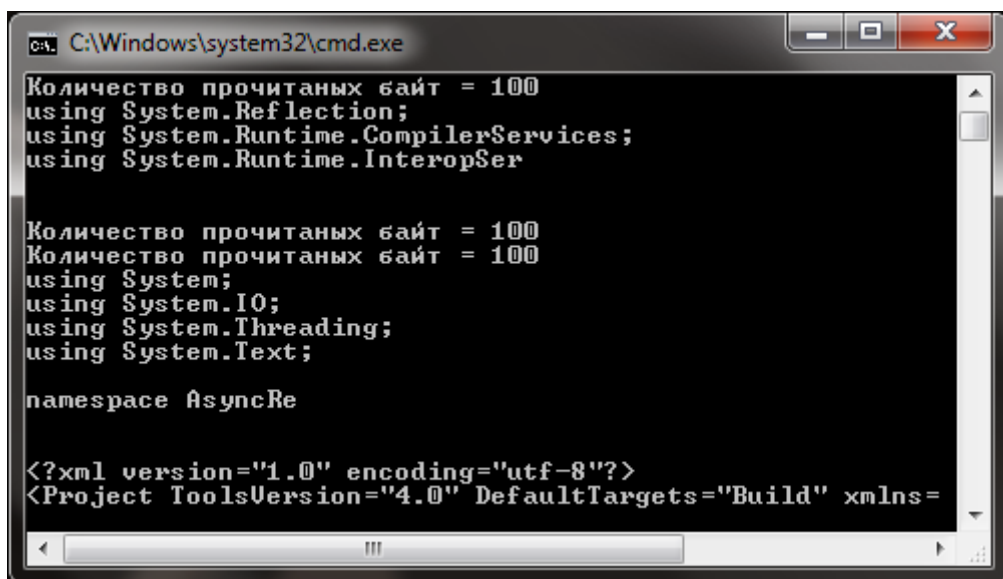
```
public delegate void AsyncBytesReadDel(Byte[] streamData);

class AsyncCallbackReader
{
    FileStream stream;
    byte[] data;
    IAsyncResult asRes;
    AsyncBytesReadDel callbackMethod;

    public AsyncCallbackReader(FileStream s, int size, AsyncBytesReadDel meth)
    {
        stream = s;
        data = new byte[size];
        callbackMethod = meth;
        asRes = s.BeginRead(data, 0, size, ReadIsComplete, null);
    }

    public void ReadIsComplete(IAsyncResult ar)
    {
        int countByte = stream.EndRead(asRes);
        stream.Close();
        Array.Resize(ref data, countByte);
        callbackMethod(data);
    }
}
```

Результат выполнения кода:



```
C:\Windows\system32\cmd.exe

Количество прочитанных байт = 100
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

Количество прочитанных байт = 100
Количество прочитанных байт = 100
using System;
using System.IO;
using System.Threading;
using System.Text;

namespace AsyncRe

<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns=
```



Итак, мы рассмотрели все возможные способы получения результата асинхронной операции. Если вам при разработке нужно будет использовать асинхронную операцию ввода-вывода – используйте методы *BeginXXX* и *EndXXX* стандартных классов (у которых эти методы есть).

Но что делать, если разработчик создал свой собственный тип, в котором хочет асинхронно вызвать собственноручно созданный метод? Данная задача решается ни чуть не сложнее, чем асинхронные вызовы рассмотренные выше. Единственное «но» в этом случае – это то, что в отличие от асинхронных операций ввода-вывода, вычислительные операции более ресурсоемкие и поэтому они более серьезно загружают потоки работой. Но одним вводом-выводом «сыт» не будешь.

```
public static UInt64 Sum(UInt64 n)
{
    UInt64 sum = 1;
    for (UInt64 i = 2; i < n; ++i)
        sum += i;

    return sum;
}
```

Для асинхронного вызова собственного метода нам понадобится создать делегат, у которого сигнатура совпадает с сигнатурой вызываемого метода. Рассмотрим это на примере метода, который вычисляет сумму чисел от единицы до указанного пользователем числа.

Данный метод будет достаточно трудоемким, если *n* будет большим. Создадим делегат для нашего метода:

```
private delegate UInt64 AsyncSumDel(UInt64 n);
```



Как вы знаете из базового курса, компилятор, встречая объявление делегата, преобразовывает его в объявление класса этого же делегата. Итого получаем следующий класс:

```
internal sealed class SumDelegate : MulticastDelegate
{
    public SumDelegate(object object, IntPtr method);
    public UInt64 Invoke(UInt64 n);
    public IAsyncResult BeginInvoke(long n,
                                    AsyncCallback callback, object object);
    public UInt64 EndInvoke(IAsyncResult result);
}
```

Видим, что в классе делегата присутствуют методы *BeginInvoke* и *EndInvoke*. Как мы помним методы *BeginXXX* нужны для запуска асинхронной операции, а методы *EndXXX* – для получения ее результата.

Теперь понятно, что использование делегата для выполнения вычислительной операции — простая задача, потому что она укладывается в рамки модели асинхронного программирования.

При асинхронном вызове метода с помощью делегата можно воспользоваться любым из способов получения результата: через ожидание завершения, с помощью регулярного опроса или обратного вызова метода. Собственно асинхронный вызов метода *Sum* выглядит следующим образом:

```
static void Main(string[] args)
{
    AsyncSumDel del = Sum;
    del.BeginInvoke(1000000, EndSum, del);
    Console.ReadKey();
}
```

Переменная *del* сначала инициализируется ссылкой на метод, который нужно вызвать асинхронно. Затем асинхронный вызов метода выполняется вызовом метода *BeginInvoke*. При этом CLR создает объект *IAsyncResult*,

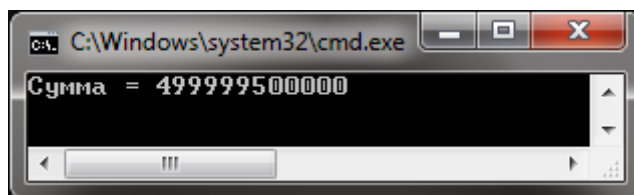
определяющий асинхронную операцию. Как уже говорилось, операции ввода-вывода ставятся в очередь драйвера устройства Windows. А метод *BeginInvoke* делегата ставит вычислительные операции в очередь пула потоков CLR, вызывая метод *QueueUserWorkItem* объекта *ThreadPool*. И, наконец, *BeginInvoke* возвращает объект *IAsyncResult* вызывающему коду. Теперь этот объект можно использовать так же, как и при выполнении асинхронной операции ввода-вывода.

После того как метод *BeginInvoke* поставит операцию в очередь в пул потоков CLR, свободный поток из пула потоков активизируется, извлекает из очереди «рабочий элемент» и вызывает *Sum*. Обычно, завершив выполнение метода, поток возвращается в пул. Но в данном примере при вызове *BeginInvoke* ему было передано в качестве предпоследнего параметра имя метода *EndSum*. Поэтому, когда *Sum* возвращает управление приложению, поток из пула не возвращается обратно в пул, а вызывает метод *EndSum*. Иначе говоря, обратный вызов происходит при завершении вычислительной операции так же, как и при завершении операции ввода-вывода. Метод *EndSum* выглядит следующим образом:

```
private static void EndSum(IAsyncResult ar)
{
    AsyncSumDel del = (AsyncSumDel)ar.AsyncState;
    UInt64 res = del.EndInvoke(ar);

    Console.WriteLine("Сумма = " + res);
}
```

В результате выполнения программы получаем следующий результат:





Итак, подведем итоги данного раздела. Модель асинхронного программирования применять не только можно, но и нужно. Эта модель довольно просто реализуема, как вы могли убедиться по представленным примерам. Ее поддерживает много классов FCL, но при необходимости можно и пользовательский метод «заставить» асинхронно выполняться.

Если вы решили придерживаться этой модели, то есть несколько нюансов, которые вам нужно знать.

Чтобы избежать утечки ресурсов, необходимо вызывать метод *EndXXX*. Делать это обязательно по двум причинам. Во-первых, при старте асинхронной операции CLR выделяет под нее внутренние ресурсы. При завершении операции CLR резервирует эти ресурсы до вызова метода *EndXXX*. Если его не вызвать, эти ресурсы освободятся лишь тогда, когда будет завершаться процесс. Во-вторых, при старте асинхронной операции нельзя предсказать, каким будет ее завершение (успешным или нет). Узнать это можно, только вызвав метод *EndXXX* и посмотрев возвращаемое значение или сгенерированное им исключение.

В любой асинхронной операции не следует вызывать *EndXXX* более одного раза. При вызове метода *EndXXX* он обращается к внутренним ресурсам, а затем освобождает их. Результат повторного вызова *EndXXX* предсказать нельзя, потому как ресурсы уже были освобождены.

Методы *BeginXXX* и *EndXXX* должны вызываться с одним и тем же объектом. То есть нельзя создать делегат и вызвать его метод *BeginInvoke*, а позже создать еще один делегат, ссылающийся на тот же метод, и использовать его для вызова *EndInvoke*. Такая программа работать не будет т.к. в объекте *IAsyncResult* хранится ссылка на исходный объект, использованный при вызове *BeginInvoke*, и если они не совпадают, *EndInvoke* сгенерирует исключение *InvalidOperationException* с сообщением о том, что объект *IAsyncResult* не соответствует делегату.



3 Использование таймеров обратного вызова

Во многих программах требуется следить за временем или выполнять какие-либо периодические действия. Программы MS-DOS для работы с таймером перехватывали аппаратное прерывание таймера, встраивая свой собственный обработчик для прерывания INT 8h. Обычные приложения Windows не могут самостоятельно обрабатывать прерывания таймера, поэтому для работы с ним нужно использовать другие способы.

Операционная система Windows позволяет для каждого приложения создать несколько виртуальных таймеров. Все эти таймеры работают по прерываниям одного физического таймера.

Так как работа Windows основана на передаче сообщений, логично было бы предположить, что и работа виртуального таймера также основана на передаче сообщений. И в самом деле, приложение может заказать для любого своего окна несколько таймеров, которые будут периодически посылать в функцию окна сообщение с кодом WM_TIMER.

В пространстве имен *System.Threading* определен класс *Timer*, с помощью которого можно периодически вызывать методы в CLR.

У класса *Timer* есть несколько похожих друг на друга конструкторов.

```
public Timer(TimerCallback callback, object state,
            int dueTime, int period);
public Timer(TimerCallback callback, object state,
            uint dueTime, int period);
public Timer(TimerCallback callback, object state,
            long dueTime, long period);
public Timer(TimerCallback callback, object state,
            TimeSpan dueTime, TimeSpan period);
```



Все четыре конструктора создают объект *Timer*. Параметр *callback* содержит имя метода, обратный вызов которого должен выполняться потоком из пула.

Конечно, созданный метод обратного вызова должен соответствовать типу-делегату *System.Threading.TimerCallback*:

```
delegate void TimerCallback(object state);
```

Параметр *state* конструктора нужен для передачи методу обратного вызова данных о состоянии или *null*, если таких данных нет. Параметр *dueTime* позволяет задать для CLR время ожидания (в мс) перед первым вызовом метода обратного вызова. Если метод обратного вызова должен вызываться немедленно, то нужно указать в этом параметре 0.

Последний параметр, *period*, указывает периодичность (в мс) вызовов метода обратного вызова. Если ему передано *Timeout.Infinite*, метод обратного вызова вызовется из пула потоков только один раз.

В CLR существует только один поток, который используется для всех объектов *Timer*. Именно он и знает о времени следующего объекта *Timer*. Когда приходит время следующего объекта *Timer*, поток CLR просыпается, вызывает метод *QueueUserWorkItem* объекта *ThreadPool*, чтобы добавить запись в очередь пула потоков для вызова метода обратного вызова. Если метод обратного вызова выполняется долго, таймер может сработать опять. Вполне возможна ситуация, в которой один метод обратного вызова выполняется несколькими потоками из пула. Здесь нужно быть очень внимательным: если метод обращается к совместно используемым ресурсам, то лучше добавить блокировки синхронизации потоков, чтобы защитить эти данные от повреждения.



У класса *Timer* есть несколько дополнительных методов, позволяющих указать для CLR время и условия обратного вызова метода. Это методы *Change* и *Dispose*:

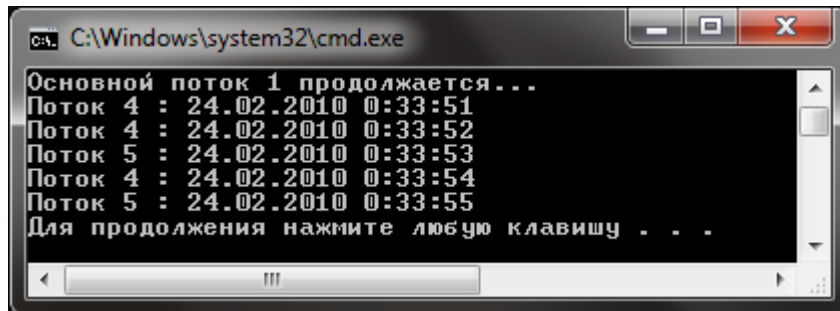
```
public bool Change(int dueTime, int period);  
public bool Change(uint dueTime, uint period);  
public bool Change(long dueTime, long period);  
public bool Change(TimeSpan dueTime, TimeSpan period);  
public bool Dispose();  
public bool Dispose(WaitHandle notifyObject);
```

С помощью метода *Change* можно изменить или сбросить значение счетчика *Timer*. Метод *Dispose* — полностью отключает таймер или сообщает объекту ядра, указанному в параметре *notifyObject*, когда все ожидающие обратные вызовы будут завершены.

Рассмотрим простой пример, где поток таймера выводит на консоль текущую дату и время:

```
using System;  
using System.Threading;  
  
namespace TimerTest  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Timer t = new Timer(TimerMethod, null, 0, 1000);  
  
            Console.WriteLine("Основной поток {0} продолжается...",  
                             Thread.CurrentThread.ManagedThreadId);  
            Thread.Sleep(5000);  
            t.Dispose();  
        }  
  
        static void TimerMethod(Object obj)  
        {  
            Console.WriteLine("Поток {0} : {1}", Thread.CurrentThread.ManagedThreadId,  
                             DateTime.Now);  
        }  
    }  
}
```

Результат выполнения этого кода:



```
C:\Windows\system32\cmd.exe
Основной поток 1 продолжается...
Поток 4 : 24.02.2010 0:33:51
Поток 4 : 24.02.2010 0:33:52
Поток 5 : 24.02.2010 0:33:53
Поток 4 : 24.02.2010 0:33:54
Поток 5 : 24.02.2010 0:33:55
Для продолжения нажмите любую клавишу . . .
```

Всего в библиотеке FCL существует три вида таймеров:

- Класс `Timer` из пространства имен `System.Threading` (который мы рассматривали выше). Он лучше других подходит для выполнения периодических фоновых задач в другом потоке.

- Класс `Timer` из пространства имен `System.Windows.Forms`. Когда запущен этот таймер, Windows добавляет в очередь сообщений потока сообщение таймера (`WM_TIMER`). Поток выбирает эти сообщения из очереди сообщений и передает их нужному методу обратного вызова. При этом установка таймера и обработка метода обратного вызова выполняются одним и тем же потоком. Это предотвращает параллельное выполнение метода таймера несколькими потоками.

- Класс `Timer` из пространства имен `System.Timers` является оболочкой для класса `Timer` из пространства имен `System.Threading`. Он происходит от класса `Component` из пространства имен `System.ComponentModel`, что дает возможность разработчикам размещать объекты-таймеры в области конструктора форм в Microsoft Visual Studio.

4 Проблемы синхронизации.

Если потоки приложений не ожидают завершения операций, то приложения будут отличаться высокой производительностью. Именно поэтому лучше всего реализовывать методы, работающие с собственными данными, и избегать методов, обращающихся к каким-либо совместно используемым данным. Тем не менее, такие ситуации, когда поток работает только со своими данными и не обращается к общим – бывает редко.

Даже если вам удастся организовать отдельный доступ потоков к переменным, то все равно рано или поздно потокам придется обращаться к общим системным ресурсам: кучам, последовательным портам, файлам, окнам и многому другому. Если один поток запрашивает исключительный доступ к ресурсу, остальные потоки, которым тоже нужен этот ресурс, не смогут завершить свою работу. С другой стороны, нельзя просто разрешать любому потоку работать с любым ресурсом в любое время.

Если игнорировать это требование, то когда-нибудь случится ситуация, когда один поток попытается сделать запись в блок памяти, когда другой поток будет считывать эту информацию.

Чтобы предотвратить повреждение ресурса по причине одновременного доступа многих потоков, нужно использовать конструкции синхронизации потоков.

В Windows и CLR таких конструкций много, и у каждой свои преимущества и недостатки. Далее мы подробнее познакомимся с этими нюансами.

Многие из конструкций синхронизации потоков в CLR на самом деле всего лишь оболочки классов, построенные на базе конструкций синхронизации потоков Win32. От этого никуда не деться, так как потоки



CLR — это потоки Windows, а это значит, что именно Windows планирует и контролирует их синхронизацию.

Давайте сейчас рассмотрим проблемную задачу, которую невозможно решить без использования синхронизации.

Создадим класс *Counter* со статическим (для простоты сделаем его общедоступным) полем *count*. Именно это поле будет общим ресурсом для всех потоков.

```
class Counter
{
    public static int count;
}
```

Теперь в основном методе создадим пять потоков, которые будут вызывать анонимный метод по увеличению поля *count* на 1000000. Итого после отработки всех потоков значение поля *count* должно быть 5000000.

```
static void Main(string[] args)
{
    Thread[] threads = new Thread[5];

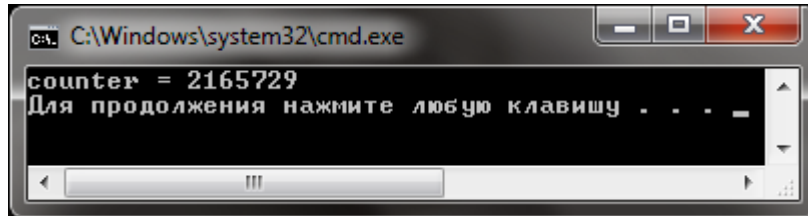
    for (int i = 0; i < threads.Length; ++i)
    {
        threads[i] = new Thread(delegate()
        {
            for (int j = 1; j <= 1000000; ++j)
                ++Counter.count;
        });

        threads[i].Start();
    }

    for (int i = 0; i < threads.Length; ++i)
        threads[i].Join();

    Console.WriteLine("counter = {0}", Counter.count);
}
```

В результате выполнения данной программы у меня на экране отобразилось следующее (хотя результат может отличаться):



В однопроцессорных (одноядерных) системах результат будет получаться правильным. Но при выполнении этого кода разными процессорами точного результата без применения синхронизации вы не получите.

Это происходит из-за механизма обновления поля *count* процессором. Дело в том, что обычная операция увеличения поля на единицу происходит в три этапа:

- Считывание значения в регистр процессора
- Увеличение значения на единицу
- Копирование значения из регистра в память

И поскольку эта операция выполняется не атомарно сразу несколькими потоками, то планировщик потоков может отключить любой из них по истечению времени (20 мс.) и передать управление другому. Предыдущий поток в это время не успевает досчитать значение или поместить его обратно в память. Рассмотрим первую итерацию такого цикла, выполняемую одновременно двумя потоками:

- Поток 1 считывает *count* в регистр → 0
- Поток 1 увеличивает значение регистра → 1
- Планировщик отключает Поток 1



- Планировщик подключает Поток 2
- Поток 2 считывает *count* в регистр → 0
- Поток 2 увеличивает значение регистра → 1
- Поток 2 сохраняет значение в память → 1
- Планировщик отключает Поток 2
- Планировщик подключает Поток 1
- Поток 1 сохраняет значение в память → 1

Это только одна из возможных ситуаций, но уже видно, что за одну итерацию цикла в каждом из двух потоков мы потеряли одно значение. Соответственно за миллион итераций в каждом потоке потерянных значений будет больше.

Таким образом, без синхронизации работы потоков правильный результат мы не получим никогда.

Синхронизацию можно проводить любым из способов, которые будут рассмотрены далее в уроке.

5 Взаимоисключающий доступ. Класс *Interlocked*.

Самый быстрый способ обеспечить взаимоисключающий доступ нескольких потоков к общему ресурсу — задействовать семейство *Interlocked*-методов. По сравнению с другими конструкциями синхронизации эти методы работают очень быстро и к тому же просты в применении.

Все методы являются статическими и находятся внутри класса *Interlocked*:



```
public static Int32 Increment(ref Int32 location); —
```

увеличивает значение на 1;

```
public static int Decrement(ref int location); —
```

уменьшает значение на 1;

```
public static int Add(ref int location, int value); —
```

увеличивает/уменьшает значение на value;

```
public static int Exchange(ref int locationi, int value); —
```

обменивает параметры значениями;

```
public static int CompareExchange(ref int location, int value,  
int comparand) —
```

сравнивает location и comparand и присваивает location value в случае успеха.

Давайте исправим пример, который рассматривался в предыдущем разделе, с использованием Interlocked-методов.

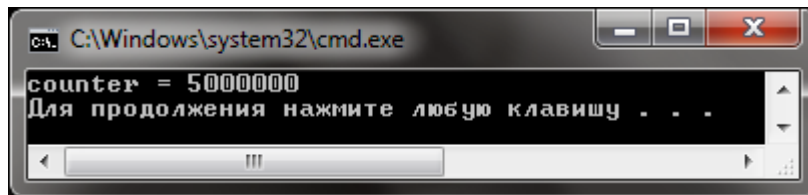
```
static void Main(string[] args)
{
    Thread[] threads = new Thread[5];

    for (int i = 0; i < threads.Length; ++i)
    {
        threads[i] = new Thread(delegate()
        {
            for (int j = 1; j <= 1000000; ++j)
                Interlocked.Increment(ref Counter.count);
        });
        threads[i].Start();
    }

    for (int i = 0; i < threads.Length; ++i)
        threads[i].Join();

    Console.WriteLine("counter = {0}", Counter.count);
}
```

В этом случае результат уже будет правильным:



6 Критическая секция.

Надеюсь, что из курса системного программирования Win32 вы помните такое понятие как критическая секция. Это специальная структура (*CRITICAL_SECTION*), которая сопоставлялась с каждым объектом в куче и с помощью определенных методов (*InitializeCriticalSection*, *DeleteCriticalSection*, *EnterCriticalSection* и *LeaveCriticalSection*) осуществлялась взаимоисключающая блокировка таких объектов.

CLR не предоставляет такой структуры, но предоставляет для каждого объекта отдельное поле по типу поля *CRITICAL_SECTION*, а также берет на себя задачи инициализации и удаления этого поля.

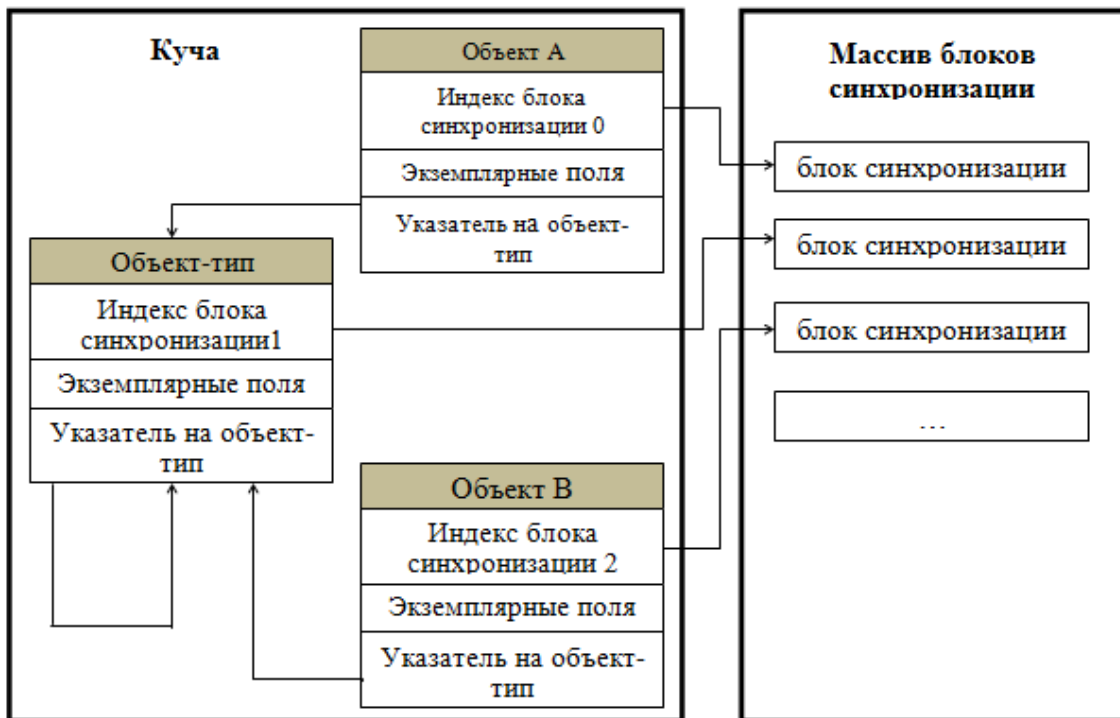
На самом деле, сопоставление поля *CRITICAL_SECTION* (которое занимает до сорока байт) с каждым объектом в куче довольно расточительно, особенно если для большинства объектов никогда не требуется безопасный доступ. Чтобы снизить расходование памяти, был придуман более эффективный способ предоставления только что описанной функциональности. При инициализации CLR выделяется память под массив участков памяти, которые можно связать с объектом. Эти участки называют



блоками синхронизации. Каждый блок синхронизации содержит те же поля, что структура *CRITICAL_SECTION*.

Когда любой объект размещается в куче, помимо его данных, создаются два дополнительных служебных поля. Первое поле – это указатель на «объект-тип» нашего типа. Второе поле – содержит целочисленный индекс блока синхронизации в массиве блоков синхронизации.

При создании объекта индексу блока синхронизации присваивается отрицательное значение, которое указывает на то, что нет блока синхронизации, сопоставленного объекту. После этого, когда вызывается метод для входа в блок синхронизации объекта, CLR находит в массиве свободный блок синхронизации и задает индексу блока синхронизации объекта значение, соответствующее найденному блоку. Таким образом, блоки синхронизации связываются с объектами прямо во время выполнения приложения. После того как все потоки освобождают блок синхронизации объекта, индекс блока синхронизации опять устанавливается в отрицательное значение, чтобы считаться свободным, и может связываться с другим объектом.



Логически у каждого объекта в куче есть связанный с ним блок синхронизации, который можно использовать для быстрой синхронизации потоков. Однако физически структуры блоков синхронизации связываются с объектом только по необходимости, а когда эта необходимость отпадает, привязка снимается. Это обеспечивает эффективность использования памяти. Кстати, при необходимости в массиве блоков синхронизации могут создаваться дополнительные блоки, поэтому не стоит беспокоиться, что системе может не хватить блоков, если потребуется синхронизировать много объектов.

6.1 Класс Monitor.

Так как напрямую работать с критической секцией мы не можем, то был придуман класс *System.Threading.Monitor*. Данный класс инкапсулирует критическую секцию и берет всю работу по ее созданию, инициализации, уничтожению и использованию. Чтобы заблокировать или разблокировать



блок синхронизации, нужно вызывать статические методы, определенные в классе *Monitor*. Для блокировки объекта нужно вызвать метод:

```
static void Enter(object obj);
```

При вызове этот метод сначала проверяет, отрицателен ли указанный индекс блока синхронизации; если да, метод находит свободный блок синхронизации и записывает его индекс в индекс блока синхронизации объекта. После связывания блока с объектом метод *Monitor.Enter* проверяет указанный блок синхронизации — не занят ли он другим потоком. Если блок свободен, вызывающий поток становится владельцем блока.

Если же этим блоком владеет другой поток, вызывающий поток приостанавливается, пока владеющий этим блоком поток не освободит его.

Также можно использовать любую перегрузку метода *Monitor.TryEnter*, который не приостанавливает поток, если объект заблокирован, а возвращает *false*:

```
static bool TryEnter(object obj);
```

```
static bool TryEnter(object obj, int millisecondsTimeout);
```

```
static bool TryEnter(object obj, TimeSpan timeout);
```

Первая перегрузка проверяет, может ли вызывающий поток занять блок синхронизации и при положительном результате возвращает значение *true*. Другие версии позволяют указать время тайм-аута — сколько вызывающий поток будет ожидать освобождения блока. Все методы возвращают *false*, если занять блок синхронизации не удалось.



После того как поток стал владельцем блока синхронизации, код получает доступ к любым данным блока синхронизации, для защиты которых он используется. По завершении работы с блоком поток должен освободить его вызывая метод *MonitorExit*:

```
static void Exit(object obj);
```

Если поток, вызывающий метод *MonitorExit*, не владеет указанным блоком синхронизации, то генерируется исключение *SynchronizationLockException*.

Модифицируем пример из предыдущей главы так, чтобы в классе Counter было два целочисленных поля. Каждый поток увеличивает первое поле на единицу, а второе увеличивает на единицу, только если первое поле имеет четное значение. Для начала рассмотрим класс *InterlockedCounter*, который использует синхронизацию с помощью Interlocked-методов. Данный подход не сработает, поскольку между двумя операциями инкрементирования планировщик потоков может приостановить один и запустить другой. И поэтому значения могут быть считаны не правильно.

Также мы напишем класс *MonitorLockCounter* который для синхронизации использует класс Monitor. Этот класс правильный, так как обе операции инкрементирования выполняются как одна атомарная.



```
class InterlockedCounter
{
    int field1;
    int field2;

    public int Field1
    {
        get { return field1; }
    }

    public int Field2
    {
        get { return field2; }
    }

    public void UpdateFields()
    {
        for (int i = 0; i < 1000000; ++i)
        {
            Interlocked.Increment(ref field1);
            if (field1 % 2 == 0)
                Interlocked.Increment(ref field2);
        }
    }
}
```

А теперь – метод где используется класс *InterlockedCounter*:



```
private static void BadAsync()
{
    Console.WriteLine("Синхронизация Interlocked-методами:");
    InterlockedCounter c = new InterlockedCounter();

    Thread[] threads = new Thread[5];
    for (int i = 0; i < threads.Length; ++i)
    {
        threads[i] = new Thread(c.UpdateFields);
        threads[i].Start();
    }

    for (int i = 0; i < threads.Length; ++i)
        threads[i].Join();

    Console.WriteLine("Field1: {0}, Field2: {1}\n\n", c.Field1, c.Field2);
}
```

А теперь напомним класс, использующий методы класса *Monitor*:

```
class MonitorLockCounter
{
    int field1;
    int field2;

    public int Field1 { get { return field1; }}
    public int Field2 { get { return field2; }}

    public void UpdateFields()
    {
        for (int i = 0; i < 1000000; ++i)
        {
            Monitor.Enter(this);
            try
            {
                ++field1;
                if (field1 % 2 == 0)
                    ++field2;
            }
            finally
            {
                Monitor.Exit(this);
            }
        }
    }
}
```

И теперь пример метода, который использует этот класс:

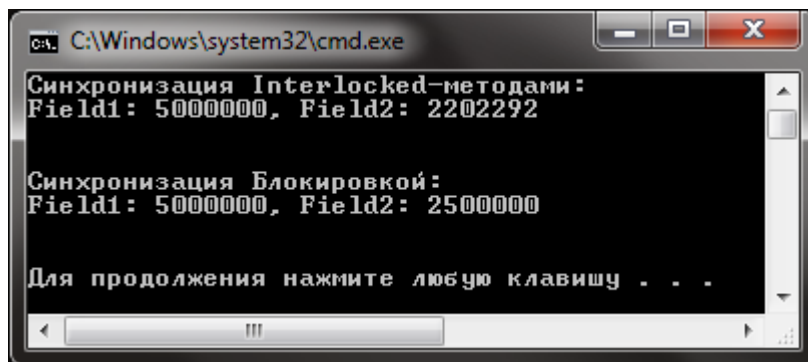
```
private static void GoodAsync()
{
    Console.WriteLine("Синхронизация Блокировкой:");
    MonitorLockCounter c = new MonitorLockCounter();

    Thread[] threads = new Thread[5];
    for (int i = 0; i < threads.Length; ++i)
    {
        threads[i] = new Thread(c.UpdateFields);
        threads[i].Start();
    }

    for (int i = 0; i < threads.Length; ++i)
        threads[i].Join();

    Console.WriteLine("Field1: {0}, Field2: {1}\n\n", c.Field1, c.Field2);
}
```

Результат выполнения:



```
C:\Windows\system32\cmd.exe
Синхронизация Interlocked-методами:
Field1: 50000000, Field2: 2202292

Синхронизация Блокировкой:
Field1: 50000000, Field2: 25000000

Для продолжения нажмите любую клавишу . . .
```

6.2 Ключевое слово lock.

Поскольку последовательность: вызовов метода `Monitor.Enter`, использование ресурса и вызов метода `Monitor.Exit` используется очень часто – была придумана сокращенная форма записи при помощи ключевого слова `lock`. Синтаксис записи следующий:



```
public void SomeMethod()  
{  
    lock(this)  
    {  
        Использование ресурса  
    }  
}
```

Таким образом, задача по инкрементированию полей из предыдущего примера – упрощается:

```
public void UpdateFields()  
{  
    for (int i = 0; i < 1000000; ++i)  
    {  
        lock(this);  
        {  
            ++field1;  
            if (field1 % 2 == 0)  
                ++field2;  
        }  
    }  
}
```

Использование оператора `lock` явно упрощает код и при этом обеспечивает вызов метода `Monitor.Exit` в конце блока, даже если возникнет исключительная ситуация (при возникновении которой без применения оператора `lock` необходимо было задействовать секцию `finally` для освобождения объекта).

Все хорошо, пока у нас есть объект в куче. Но как быть, если мы используем статический класс и хотим обеспечить исключительный доступ через статический метод? Объекта в куче нет и поэтому нет блока индекса синхронизации. В этом случае вспоминаем, что у каждого объекта в куче

есть указатель на его «объект-тип». Поскольку данный объект находится в куче, то у него имеется два служебных поля: индекс блока синхронизации и указатель на объект-тип, а значит, его можно использовать для блокировки. Т.е. ссылку на «объект-тип» можно передавать методам – *Enter*, *TryEnter* и *Exit*.

Изменим слегка предыдущий пример для демонстрации синхронизации с блокировкой статического типа. Для этого изменим объявление класса и всего его содержимого как статического.

```
static class LockCounter
{
    static int field1;
    static int field2;

    public static int Field1
    {
        get { return field1; }
    }

    public static int Field2
    {
        get { return field2; }
    }

    public static void UpdateFields()
    {
        for (int i = 0; i < 1000000; ++i)
        {
            lock(typeof(StaticLockCounter))
            {
                ++field1;
                if (field1 % 2 == 0)
                    ++field2;
            }
        }
    }
}
```

Использование и результаты остаются такими же, как и в предыдущем примере.



Синхронизация с помощью блокировок проста в использовании и оптимизирована по быстродействию, но как любят говорить в Microsoft – «... имеет уязвимость, которая позволяет злоумышленнику...» вызвать взаимную блокировку потоков использующих данный объект.

Проектируя механизм синхронизации, специалисты из Microsoft почему-то не учли тот факт, что блок синхронизации имеет открытую структуру синхронизации данных, связанную с каждым объектом кучи! Таким образом, любой код, имеющий ссылку на объект, в любой момент может передать эту ссылку в методы *Enter* и *Exit* и перехватить блокировку. Более того – любой код может передать в эти методы ссылку на любой «объект-тип» и перехватить блокировку этого типа.

Рассмотрим код, который описывает ситуацию взаимной блокировки. Это происходит тогда, когда основной поток блокирует объект, и в это время иницируется сборка мусора. При этом метод *Finalize* объекта *SomeType*, пытается перехватить блокировку объекта для каких-либо нужд. Но завершающий поток CLR не может получить блокировку объекта, так как ею владеет основной поток приложения. Это приводит к остановке потока финализатора вследствие чего ни один из процесс не сможет завершиться и следовательно никакая память, занятая объектами в куче, не может освободиться.

Результаты выполнения программы не привожу т.к. на экране ничего не отобразится в результате взаимной блокировки потоков. Более того – приложение не закроется самостоятельно.



```
using System;
using System.Threading;

namespace DeadLock
{
    class DeadLockClass
    {
        static void Main(string[] args)
        {
            DeadLockClass dlc = new DeadLockClass();

            Monitor.Enter(dlc);
            dlc = null;

            GC.Collect(0);
            GC.WaitForPendingFinalizers();

            Console.WriteLine("Эта строка не будет напечатана");
        }

        ~DeadLockClass()
        {
            lock (this)
            {
                Console.WriteLine("Запись лог-файла");
            }
        }
    }
}
```

Для решения проблемы открытой блокировки просто нужно создать закрытое поле типа *System.Object* и передать его оператору *lock* либо методам *Enter* и *Exit*.

Продемонстрируем корректную версию синхронизации с использованием блокировки на примере класса счетчика из предыдущего примера. Даже при блокировке самого объекта метод увеличения полей работает корректно, т.к. внутри блокируется закрытое поле *lockObj*, а не сам объект. И получить доступ к этому полю извне – нельзя.



```
using System;
using System;
using System.Threading;

namespace GoodLockSync
{
    class GoodLockSyncClass
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Синхронизация статического типа:");
            LockCounter lc = new LockCounter();
            Monitor.Enter(lc);
            Thread[] threads = new Thread[5];
            for (int i = 0; i < threads.Length; ++i)
            {
                threads[i] = new Thread(lc.UpdateFields);
                threads[i].Start();
            }

            for (int i = 0; i < threads.Length; ++i)
                threads[i].Join();

            Console.WriteLine("Count: {0}\n\n", lc.Count);
        }
    }

    class LockCounter
    {
        int count;
        object lockObj = new object();

        public int Count
        {
            get { return count; }
        }

        public void UpdateFields()
        {
            for (int i = 0; i < 1000000; ++i)
            {
                lock (lockObj)
                {
                    ++count;
                }
            }
        }
    }
}
```



7 Мьютексы. Класс `Mutex`.

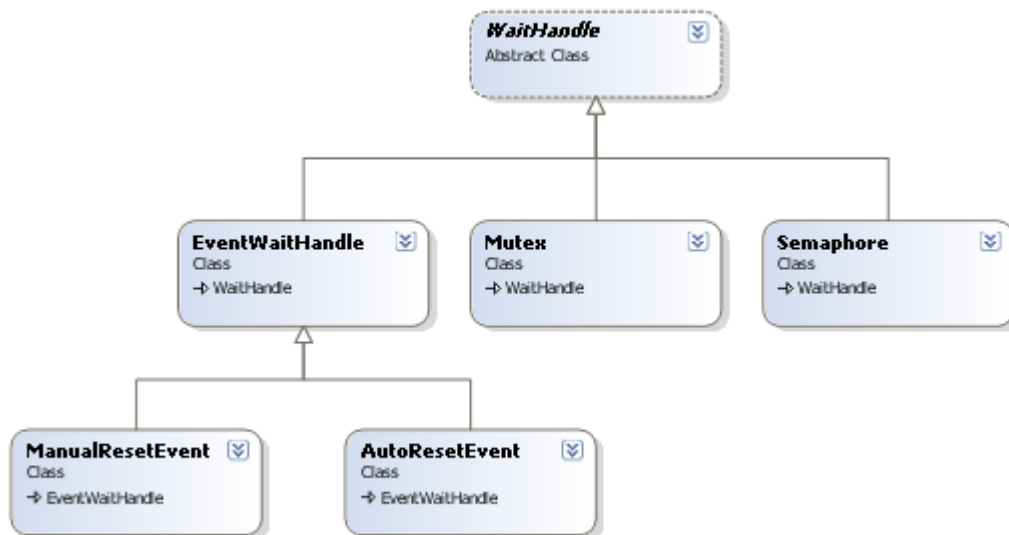
Windows предлагает несколько объектов ядра для выполнения синхронизации потоков: мьютексы, семафоры и события. В данной главе мы рассмотрим базовый класс для всех объектов синхронизации, а затем – работу объекта мьютекс, и случаи, когда его нужно использовать.

Возникает вопрос: зачем использовать синхронизацию с помощью объектов ядра, если мы уже рассмотрели варианты синхронизации с помощью `Interlocked`-методов и блокировок? Дело в том, что методы класса `Monitor` и `Interlocked`-методы позволяют выполнять синхронизацию потоков одного домена `AppDomain`, а объекты ядра можно использовать для синхронизации потоков различных `AppDomain` или процессов.

Соответственно теперь возникает вопрос: а почему бы тогда не отказаться от других способов синхронизации в пользу объектов ядра? На самом деле ожидая освобождения объекта ядра, поток должен переходить из пользовательского режима в режим ядра, что сильно снижает производительность. Поэтому синхронизация с использованием объектов ядра — самый медленный из механизмов синхронизации. И работает он примерно в 30 раз медленнее применения методов класса `Monitor`. Так что вам решать в каких случаях, какие способы синхронизации применять.

В пространстве имен `System.Threading` есть абстрактный класс `WaitHandle`. Этот класс служит оболочкой описателя объекта ядра. Именно от него наследуются все классы объектов ядра, которые используются для синхронизации.

Иерархия классов наследников *WaitHandle* выглядит следующим образом:



Все классы-потомки наследуют от *WaitHandle* следующие методы:

- **`public virtual void Close();`** – закрывает дескриптор объекта ядра;
- **`public virtual bool WaitOne();`** – ожидает сигнального состояния одного объекта ядра;
- **`public virtual bool WaitOne(int millisecondsTimeout, bool exitContext);`** – ожидает сигнального состояния одного объекта ядра пока не истечет таймаут;
- **`public static int WaitAny(WaitHandle[] waitHandles);`** – ожидает сигнального состояния любого объекта ядра из массива;
- **`public static int WaitAny(WaitHandle[] waitHandles, int millisecondsTimeout, bool exitContext);`** – ожидает сигнального состояния любого объекта ядра из массива пока не истечет таймаут;



- `public static bool WaitAll(WaitHandle[] waitHandles);` – ожидает сигнального состояния всех объектов ядра из массива;
- `public static bool WaitAll(WaitHandle[] waitHandles, int millisecondsTimeout, bool exitContext);` – ожидает сигнального состояния всех объектов ядра из массива пока не истечет таймаут;
- `public static bool SignalAndWait(WaitHandle toSignal, WaitHandle toWaitOn);` – переводит объект ядра в сигнальное состояние и ждет сигнального состояния от другого объекта ядра;
- `public static bool SignalAndWait(WaitHandle toSignal, WaitHandle toWaitOn, int millisecondsTimeout, bool exitContext);` – переводит объект ядра в сигнальное состояние и ждет сигнального состояния от другого объекта ядра.

Итого, вернемся к объекту класса `Mutex`. Мьютекс – переводится как «взаимное исключение». Это один из классов, который позволяет организовать синхронизацию в множестве процессов. Данный класс похож на класс `Monitor` тем, что допускает наличие только одного владельца. То есть только один поток может заблокировать мьютекс и получить доступ к общим ресурсам.

Создавая мьютекс, мы можем указать с помощью параметра `initiallyOwned`, должен ли поток-создатель сразу же заблокировать объект. Так же дополнительно можем указать имя объекта синхронизации, по которому его может открыть другой поток/процесс. А также с помощью параметра `createdNew` можем узнать, существует ли уже такой объект. Другой способ проверить существует ли уже объект-мьютекс – вызвать метод `OpenExisting`, который либо возвращает существующий объект, либо генерирует исключение `WaitHandleCannotBeOpenedException`.



Для получения блокировки мьютекса нужно вызвать из-под него метод *WaitOne*. А для освобождения – метод *ReleaseMutex*. Для демонстрации работы мьютекса – возьмем все тот же пример с классом-счетчиком.

```
using System;
using System.Threading;

namespace MutexSync
{
    class MutexSyncClass
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Синхронизация мьютексом:");
            Counter c = new Counter();

            Thread[] threads = new Thread[5];
            for (int i = 0; i < threads.Length; ++i)
            {
                threads[i] = new Thread(c.UpdateFields);
                threads[i].Start();
            }

            for (int i = 0; i < threads.Length; ++i)
                threads[i].Join();

            Console.WriteLine("Count: {0}\n\n", c.Count);
        }
    }

    class LockCounter
    {
        int count;
        object lockObj = new object();

        public int Count
        {
            get { return count; }
        }

        public void UpdateFields()
        {
            for (int i = 0; i < 1000000; ++i)
            {
                lock (lockObj)
                {
                    ++count;
                }
            }
        }
    }
}
```




Результат выполнения будет таким же, как и в прошлый раз, но подождать его придётся секунд 30 (я предупреждал, что синхронизация через объекты ядра очень медленная). Еще раз повторяюсь, что использовать мьютекс следует лишь тогда, когда необходимо организовать синхронизацию на уровне процессов, иначе всегда следует использовать синхронизацию блокировками.

8 Семафоры. Класс **Semaphore**.

Объект-семафор идентичен объекту-мьютексу, за исключением того, что семафор может использоваться множеством потоков одновременно. Внутри семафора есть счетчик, который определяет скольким потокам можно обращаться к общему ресурсу.

Создавая семафор, в параметре *initialCount* можно задать количество «доступных мест» для потоков. А в параметре *maximumCount* задается, сколько всего потоков могут одновременно работать с общим ресурсом. Остальные параметры – такие же, как и у мьютекса. Захват блокировки происходит с помощью *Wait*-методов, освобождение – с помощью метода *Release*.

Рассмотрим пример: создадим семафор, который обеспечивает трем потокам одновременный доступ к ресурсу. Каждый поток, получивший блокировку, приостанавливается на 2 секунды, после чего снимает блокировку с семафора. При попытке потока получить блокировку – установлен таймаут в 500 мс. Если семафор в течение этого времени не освобождается, то выводится соответствующее сообщение и попытки продолжаются. В основном потоке через пул потоков создаются 6 потоков, которые начинают «состязание» за ресурсы.



```
using System;
using System.Threading;

namespace SemaphoreSync
{
    class Program
    {
        static void Main(string[] args)
        {
            Semaphore s = new Semaphore(3, 3, "My_SEMAPHORE");

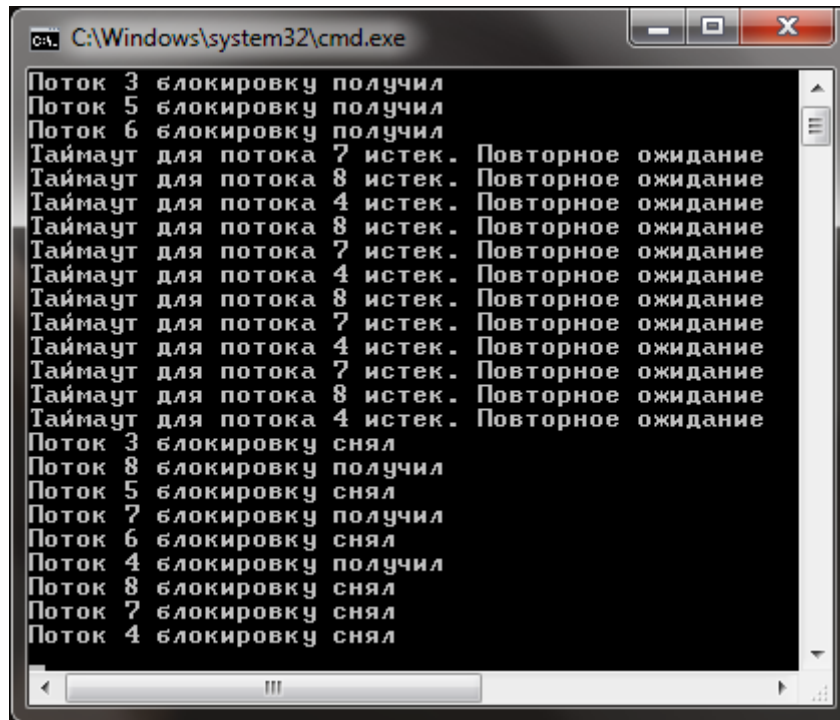
            for (int i = 0; i < 6; ++i)
                ThreadPool.QueueUserWorkItem(SomeMethod, s);

            Console.ReadKey();
        }

        static void SomeMethod(object obj)
        {
            Semaphore s = obj as Semaphore;
            bool stop = false;

            while (!stop)
            {
                if (s.WaitOne(500))
                {
                    try
                    {
                        Console.WriteLine("Поток {0} блокировку получил",
                                           Thread.CurrentThread.ManagedThreadId);
                        Thread.Sleep(2000);
                    }
                    finally
                    {
                        stop = true;
                        s.Release();
                        Console.WriteLine("Поток {0} блокировку снял",
                                           Thread.CurrentThread.ManagedThreadId);
                    }
                }
                else
                {
                    Console.WriteLine("Таймаут для потока {0} истек. Повторное ожидание", Thread.CurrentThread.ManagedThreadId);
                }
            }
        }
    }
}
```

В результате выполнения получаем следующие результаты:



```
C:\Windows\system32\cmd.exe
Поток 3 блокировку получил
Поток 5 блокировку получил
Поток 6 блокировку получил
Таймаут для потока 7 истек. Повторное ожидание
Таймаут для потока 8 истек. Повторное ожидание
Таймаут для потока 4 истек. Повторное ожидание
Таймаут для потока 8 истек. Повторное ожидание
Таймаут для потока 7 истек. Повторное ожидание
Таймаут для потока 4 истек. Повторное ожидание
Таймаут для потока 8 истек. Повторное ожидание
Таймаут для потока 7 истек. Повторное ожидание
Таймаут для потока 4 истек. Повторное ожидание
Таймаут для потока 8 истек. Повторное ожидание
Таймаут для потока 7 истек. Повторное ожидание
Таймаут для потока 4 истек. Повторное ожидание
Поток 3 блокировку снял
Поток 8 блокировку получил
Поток 5 блокировку снял
Поток 7 блокировку получил
Поток 6 блокировку снял
Поток 4 блокировку получил
Поток 8 блокировку снял
Поток 7 блокировку снял
Поток 4 блокировку снял
```

9 События.

Еще один объект синхронизации – это события. Данный объект не имеет ничего общего с событиями, которые объявляются с помощью ключевого слова `event`. Хотя по принципу функционирования – все сходится. Эти объекты уведомляют другие потоки о том, что завершились какие-то операции, поступили новые данные, необходимо выполнить какое-либо действие и т.д. События в отличие от семафоров и мьютексов не наследуются напрямую от `WaitHandle`. У них базовый класс – `EventWaitHandle`.



Для ожидания объекта-события по-прежнему используются Wait-функции. А вот для перевода в сигнальное состояние нужно использовать метод `Set`. Для сброса сигнального состояния нужен вызов метода `Reset`.

Объекты-события бывают двух видов: события с авто сбросом и события с ручным сбросом. Давайте познакомимся с ними подробнее.

9.1 Класс `ManualResetEvent`

Класс `ManualResetEvent` отвечает за события с ручным сбросом сигнального состояния. Т.е. когда события ожидают несколько потоков, то по приходу сигнального состояния – доступ получают все, кто «успел» до вызова метода `Reset`. Таким образом, мы вручную управляем доступностью ресурса.

Рассмотрим пример, где в основном потоке 10 потоков из пула потоков пытаются получить доступ к общему ресурсу. При этом каждый поток, который получает блокировку – тут же ее снимает. Но потоки настолько быстро обрабатываются, что до того времени как объект-событие перейдет в несигнальное состояние – блокировку «ухитрятся» получить несколько потоков. «Не успевшие», по таймауту закончат свое выполнение.

```
using System;
using System.Threading;

namespace EventSync
{
    class EventSyncClass
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Событие с ручным сбросом:");
            ManualResetEvent mre = new ManualResetEvent(true);

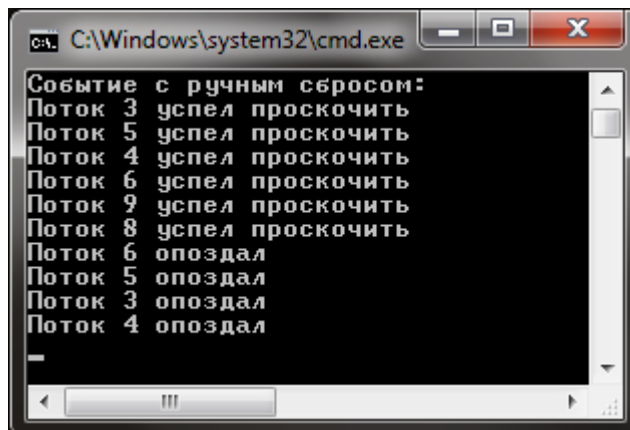
            for (int i = 0; i < 10; ++i)
                ThreadPool.QueueUserWorkItem(SomeMethod, mre);

            Console.ReadKey();
        }

        static void SomeMethod(object obj)
        {
            EventWaitHandle ev = obj as EventWaitHandle;

            if (ev.WaitOne(10))
            {
                Console.WriteLine("Поток {0} успел проскочить",
                                   Thread.CurrentThread.ManagedThreadId);
                ev.Reset();
            }
            else
                Console.WriteLine("Поток {0} опоздал",
                                   Thread.CurrentThread.ManagedThreadId);
        }
    }
}
```

В результате выполнения получим следующие результаты:

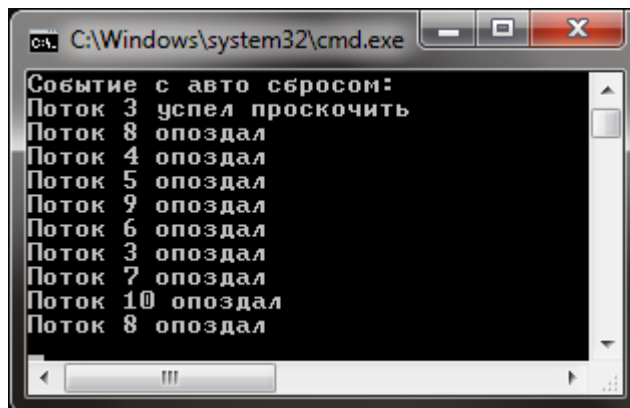


9.2 Класс `AutoResetEvent`.

Класс `AutoResetEvent` отвечает за события с автоматическим сбросом сигнального состояния. Т.е. когда события ожидают несколько потоков, то по приходу сигнального состояния – доступ получит только один поток (который автоматически сбрасывает событие в несигнальное состояние). При этом, это не тот поток, который дольше всего ждал, а тот, у которого наибольший приоритет. Таким образом, события с авто сбросом, напоминают по функциональности мьютексы.

В качестве примера рассмотрим предыдущий пример, заменив объект `ManualResetEvent` на `AutoResetEvent`. Таким образом, первый «поток-счастливчик» закроет за собой «дверь» и остальные «желающие» потоки войти не смогут.

В результате вывод на экран будет следующим:



```
C:\Windows\system32\cmd.exe
Событие с авто сбросом:
Поток 3 успел проскочить
Поток 8 опоздал
Поток 4 опоздал
Поток 5 опоздал
Поток 9 опоздал
Поток 6 опоздал
Поток 3 опоздал
Поток 7 опоздал
Поток 10 опоздал
Поток 8 опоздал
```

10 Практические примеры использования

Рассмотрим пример использования мьютекса для определения того, что приложение запущено в единственном экземпляре. Для этого пытаемся открыть именованный мьютекс при запуске приложения. Так как мьютексы



создаются на уровне операционной системы, то они видны всем процессам. Если созданного мьютекса нет, то будет сгенерировано исключение *WaitHandleCannotBeOpenedException*, а значит, приложение запускается первый раз.

```
using System;
using System.Threading;

namespace OneInstanceApp
{
    class OneInstanceAppClass
    {
        static Mutex m;

        static void Main(string[] args)
        {
            try
            {
                m = Mutex.OpenExisting("MY_MUTEX");
            }
            catch (WaitHandleCannotBeOpenedException e){}

            if (m != null)
            {
                Console.WriteLine("Приложение уже запущено!");
                Console.ReadKey();
                return;
            }

            using (m = new Mutex(true, "MY_MUTEX"))
            {
                Console.WriteLine("Приложение работает.\nНажмите любую клавишу для\nзакрытия приложения...");
                Console.ReadKey();
            }
        }
    }
}
```

```
using System;
using System.Threading;

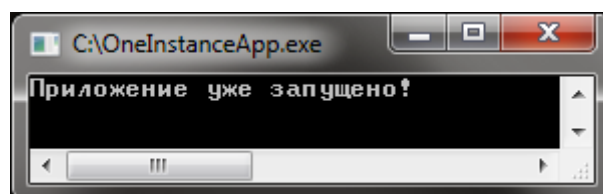
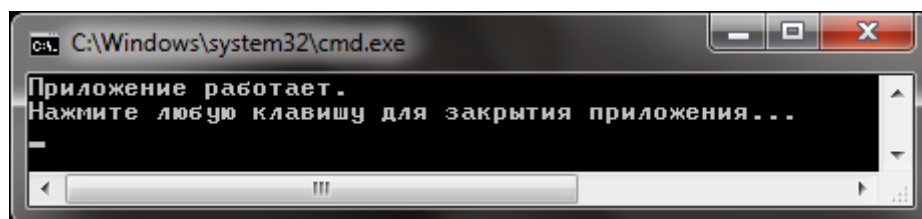
namespace OneInstanceApp
{
    class OneInstanceAppClass
    {
        static Mutex m;

        static void Main(string[] args)
        {
            try
            {
                m = Mutex.OpenExisting("MY_Mutex");
            }
            catch (WaitHandleCannotBeOpenedException e){}

            if (m != null)
            {
                Console.WriteLine("Приложение уже запущено!");
                Console.ReadKey();
                return;
            }

            using (m = new Mutex(true, "MY_Mutex"))
            {
                Console.WriteLine("Приложение работает.\nНажмите любую клавишу для\nзакрытия приложения...");
                Console.ReadKey();
            }
        }
    }
}
```

В результате запуска двух экземпляров приложения получим:





Рассмотрим пример синхронизации с помощью событий двух процессов. Задача в том, чтобы процессы по очереди распечатывали числа от нуля до десяти. Причем один процесс печатает четные, а другой – нечетные числа.

```
using System;
using System.Threading;

namespace TwoProcessEvent
{
    class TwoProcessEventClass
    {
        static EventWaitHandle canCalc;
        static int result;

        static void Main(string[] args)
        {
            try
            {
                canCalc = EventWaitHandle.OpenExisting("PROCESS_EVENT");
                canCalc.Set();
                result = 2;
            }
            catch (WaitHandleCannotBeOpenedException ex)
            {
                canCalc = new EventWaitHandle(false, EventResetMode.AutoReset,
                                                "PROCESS_EVENT");
                result = 1;
                Console.WriteLine("Запустите второй экземпляр приложения...");
            }

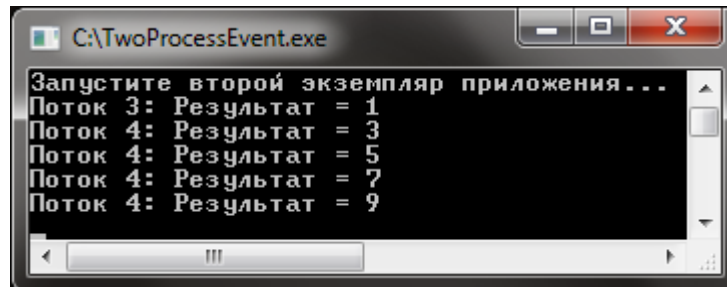
            ThreadPool.QueueUserWorkItem(ThreadMethod);

            Console.ReadKey();
        }

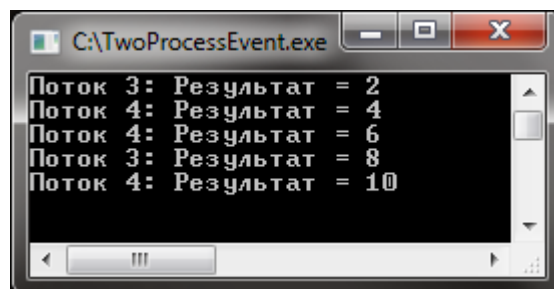
        static void ThreadMethod(object obj)
        {
            canCalc.WaitOne();
            Thread.Sleep(1000);
            Console.WriteLine("Поток {0}: Результат = {1}",
                              Thread.CurrentThread.ManagedThreadId, result);

            result += 2;
            canCalc.Set();
            Thread.Sleep(1000);
            if(result <= 10)
                ThreadPool.QueueUserWorkItem(ThreadMethod);
        }
    }
}
```

В результате, запустив два экземпляра этого приложения, получим следующий результат:



```
C:\TwoProcessEvent.exe
Запустите второй экземпляр приложения...
Поток 3: Результат = 1
Поток 4: Результат = 3
Поток 4: Результат = 5
Поток 4: Результат = 7
Поток 4: Результат = 9
```



```
C:\TwoProcessEvent.exe
Поток 3: Результат = 2
Поток 4: Результат = 4
Поток 4: Результат = 6
Поток 3: Результат = 8
Поток 4: Результат = 10
```

Теперь рассмотрим пример, в котором с помощью потоков из пула рассчитываются числа Фибоначчи. Здесь задействован массив объектов событий с авто сбросом. Несколько потоков вычисляют случайное число из последовательности чисел Фибоначчи. В это время основной поток ждет события завершения от всех «читающих» потоков. Когда данное событие поступает от всех потоков из пула, то выполнение программы завершается.

Класс *FibonacciCalculatorClass* производит вычисления, хранит результат и устанавливает событие завершения в сигнальное состояние.



```
public class FibonacciCalculatorClass
{
    private ManualResetEvent ev;

    public int Number {get; private set;}

    public int Result {get; private set;}

    public FibonacciCalculatorClass(int n, ManualResetEvent mre)
    {
        Number = n;
        ev = mre;
    }

    public void CallBackMethod(Object arg)
    {
        Console.WriteLine("Поток {0} стартовал...",
                           Thread.CurrentThread.ManagedThreadId);
        Result = CalcMethod(Number);
        Console.WriteLine("Поток {0} вычисления закончил...",
                           Thread.CurrentThread.ManagedThreadId);
        ev.Set();
    }

    public int CalcMethod(int number)
    {
        if (number <= 1)
        {
            return number;
        }

        return Calculate(number - 1) + Calculate(number - 2);
    }
}
```

В методе *Main* создается массив событий, которых будет дожидаться основной поток. Далее запускаются 10 потоков из пула и основной поток приостанавливается, ожидая завершения подсчетов.

```

class FibonacciClass
{
    static void Main()
    {
        const int threadCount = 10;

        ManualResetEvent[] finishEvents = new ManualResetEvent[threadCount];
        FibonacciCalculatorClass[] numberArr = new FibonacciCalculatorClass[threadCount];
        Random r = new Random();

        Console.WriteLine("Запуск операции подсчета...");
        for (int i = 0; i < threadCount; i++)
        {
            finishEvents[i] = new ManualResetEvent(false);
            FibonacciCalculatorClass f = new FibonacciCalculatorClass(r.Next(30, 50),
                                                                    finishEvents[i]);

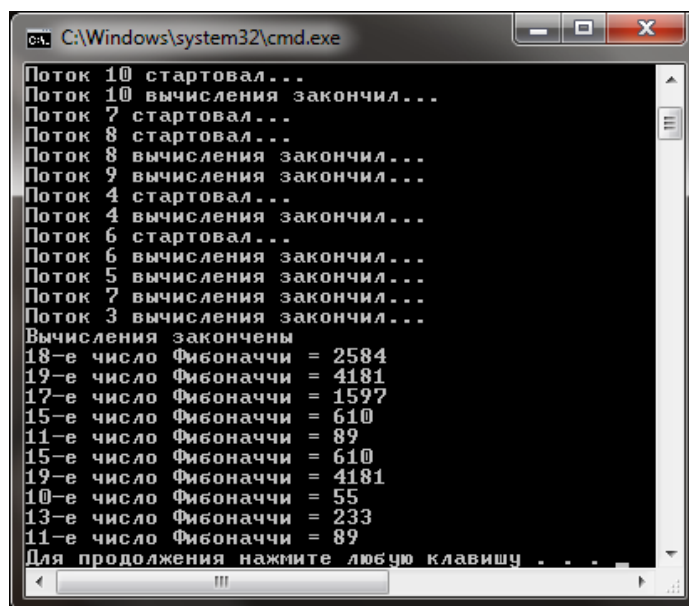
            numberArr[i] = f;
            ThreadPool.QueueUserWorkItem(f.CallBackMethod);
        }

        WaitHandle.WaitAll(finishEvents);
        Console.WriteLine("Вычисления закончены");

        for (int i = 0; i < threadCount; i++)
        {
            FibonacciCalculatorClass f = numberArr[i];
            Console.WriteLine("{0}-е число Фибоначчи = {1}", f.Number, f.Result);
        }
    }
}

```

В результате выполнения получим:



```

C:\Windows\system32\cmd.exe
Поток 10 стартовал...
Поток 10 вычисления закончил...
Поток 7 стартовал...
Поток 8 стартовал...
Поток 8 вычисления закончил...
Поток 9 вычисления закончил...
Поток 4 стартовал...
Поток 4 вычисления закончил...
Поток 6 стартовал...
Поток 6 вычисления закончил...
Поток 5 вычисления закончил...
Поток 7 вычисления закончил...
Поток 3 вычисления закончил...
Вычисления закончены
18-е число Фибоначчи = 2584
19-е число Фибоначчи = 4181
17-е число Фибоначчи = 1597
15-е число Фибоначчи = 610
11-е число Фибоначчи = 89
15-е число Фибоначчи = 610
19-е число Фибоначчи = 4181
10-е число Фибоначчи = 55
13-е число Фибоначчи = 233
11-е число Фибоначчи = 89
Для продолжения нажмите любую клавишу . . .

```



И теперь последний пример. Здесь я использовал метод класса *ThreadPool*:

```
public static RegisterWaitHandle RegisterWaitForSingleObject(WaitHandle  
    waitObject, WaitOrTimerCallback callback, object state,  
    int millisecondsTimeoutInterval, bool executeOnlyOnce);
```

Этот метод «заставляет» потоки пула потоков ждать сигнального состояния от объекта-синхронизации, либо, если истечет таймаут – продолжить выполнение. Метод использует делегат:

```
public delegate void WaitOrTimerCallback(object state, bool timedOut);
```

Параметр *timedOut* показывает, был ли метод запущен в результате истечения времени таймаута или в результате прихода сигнального состояния объекта синхронизации.

Параметр *executeOnlyOnce* указывает должен ли метод запускаться снова при переходе сигнального состояния объекта синхронизации, или отработать один раз.

А также метод:

```
public Boolean Unregister(WaitHandle waitObject);
```

служит для отказа пула потоков от «сотрудничества» с объектом синхронизации.

В данном примере показывается, как поток из пула потоков вызывает метод каждый раз, когда освобождается объект *AutoResetEvent*:

```
using System;
using System.Threading;

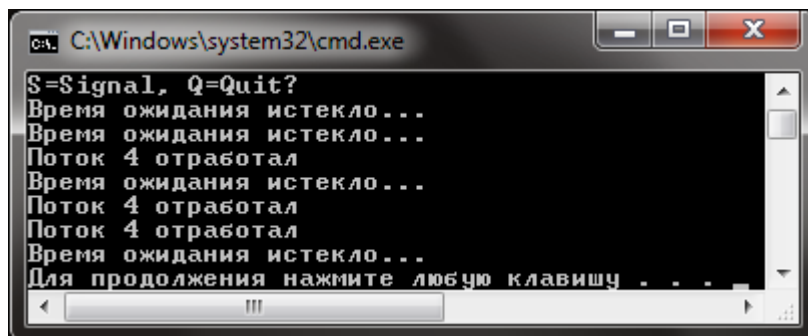
namespace PoolRegisterEvent
{
    class PoolRegisterEventClass
    {
        static void Main(string[] args)
        {
            AutoResetEvent are = new AutoResetEvent(false);
            RegisteredWaitHandle rwh = ThreadPool.RegisterWaitForSingleObject(
                are, // Ожидать этот объект AutoResetEvent.
                EventOperation, // Выполнить обратный вызов этого метода.
                null, // Передать null в качестве параметра EventOperation.
                5000, // Ждать освобождения 5 секунд.
                false); // Вызывать EventOperation при каждом освобождении.

            char operation;
            Console.WriteLine("S=Signal, Q=Quit?");
            do
            {
                operation = Char.ToUpper(Console.ReadKey(true).KeyChar);
                if (operation == 'S')
                    are.Set();
            } while (operation != 'Q');

            rwh.Unregister(null);
        }

        private static void EventOperation(Object state, Boolean timedOut)
        {
            if (timedOut)
                Console.WriteLine("Время ожидания истекло...");
            else
                Console.WriteLine("Поток {0} отработал",
                                    Thread.CurrentThread.ManagedThreadId);
        }
    }
}
```

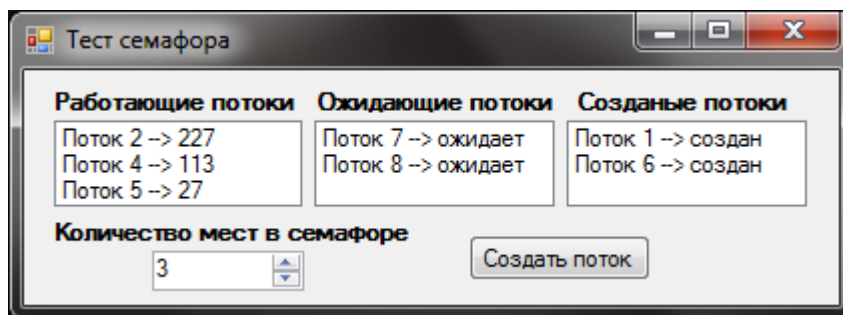
Результат выполнения следующий:



11 Домашнее задание:

1. Разработать Windows Forms приложение, которое будет использовать объект-семафор следующим образом:

По нажатию на кнопку «Создать поток» создается новый поток и помещается в первый список, где находятся все созданные потоки. Порядковый номер потока берется от 1 и увеличивается на один. При двойном клике на потоке, поток перемещается в список ожидающих потоков, где он будет находиться до тех пор, пока в семафоре не освободится для него место. Как только такое место освободилось, поток перемещается из списка ожидания в список рабочих потоков и приступает к работе. Работа заключается в том, чтобы увеличивать локальный счетчик каждого потока на единицу в секунду и отображать это значение. При двойном клике по потоку в списке рабочих потоков – поток прекращает свою работу, удаляется из списка и освобождает место для очередного ожидающего потока. Количество свободных мест задается в счетчике. При изменении счетчика более «старые» потоки покидают список, если произошло уменьшение счетчика, или же добавляются новые «ожидающие» потоки при увеличении значения счетчика. Изменение размера формы происходит, динамически подстраиваясь под наибольшее количество потоков в каком-либо списке. Примерный вид приложения смотрите ниже (хотя собственная креативность приветствуется).



2. Разработать Windows Forms приложение, которое будет имитировать мобильный телефон в плане написания SMS методом набора T9. При наборе текста окне, по нажатию клавиши должен запускаться/пробуждаться поток, который пробегая по словарю (соорудить самостоятельно) выбирает первое подходящее по сочетанию букв слово и предлагает его пользователю. Пользователь может либо согласиться, либо продолжать свой набор. При отсутствии слова в словаре и желании пользователя его туда добавить должна быть такая возможность. Постарайтесь предусмотреть смену раскладки языка (русский/английский). Тем, осилит задание сложнее – привяжите кнопки дополнительной клавиатуры к кнопкам телефона.

