МНОГОПОТОЧНОСТЬ

ОСНОВНЫЕ ВОПРОСЫ

1) Введение в многопоточность;
2) Создание потоков;
3) Потоки с параметрами;
4) Синхронизация потоков;
5) Мониторы;
6) Класс AUTORESETEVENT;
7) Мьютексы;
8) Семафоры;
9) Использование таймеров.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Одним из ключевых аспектов в современном программировании является **многопоточность**. Ключевым понятием при работе с многопоточностью является **поток**. Поток представляет некоторую часть кода программы. При выполнении программы каждому потоку выделяется определенный квант времени. И при помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи **одновременно**. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому, к примеру, клиент-серверные приложения (и не только они) практически не мыслимы без многопоточности. Таким образом, многопоточная обработка является особой формой многозадачности.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Образно многопоточность на базе потоков можно изобразить следующим образом:

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Разумеется, необходимо знать особенности одновременного выполнения множества потоков. Из-за того, что они выполняются в одно и то же время, при получении ими доступа к одним и тем же данным могут возникать проблемы. Чтобы этого не происходило, должны быть реализованы механизмы синхронизации.

Поток (thread) представляет собой независимую последовательность инструкций в программе. Потоки играют важную роль как для клиентских, так и для серверных приложений. К примеру, во время ввода какого-то кода C# в окне редактора Visual Studio проводится анализ на предмет различных синтаксических ошибок. Этот анализ осуществляется отдельным фоновым потоком. То же самое происходит и в средстве проверки орфографии в Microsoft Word. Один поток ожидает ввода данных пользователем, а другой в это время выполняет в фоновом режиме некоторый анализ. Третий поток может сохранять записываемые данные во временный файл, а четвертый — загружать дополнительные данные из Интернета.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

В приложении, которое функционирует на сервере, один поток всегда ожидает поступления запроса от клиента и потому называется потоком-слушателем (listener thread). При получении запроса он сразу же пересылает его отдельному рабочему потоку (worker thread), который дальше сам продолжает взаимодействовать с клиентом. Поток-слушатель после этого незамедлительно возвращается к своим обязанностям по ожиданию поступления следующего запроса от очередного клиента. Каждый процесс состоит из ресурсов, таких как оконные дескрипторы, файловые дескрипторы и другие объекты ядра, имеет выделенную область в виртуальной памяти и содержит как минимум один поток. Потоки планируются к выполнению операционной системой. У любого потока имеется приоритет, счетчик команд, указывающий на место в программе, где происходит обработка, и стек, в котором сохраняются локальные переменные потока. Стек у каждого потока выглядит по-своему, но память для программного кода и куча разделяются среди всех потоков, которые функционируют внутри одного процесса. Это позволяет потокам внутри одного процесса быстро взаимодействовать между собой, поскольку все потоки процесса обращаются к одной и той же виртуальной памяти. Однако это также и усложняет дело, поскольку дает возможность множеству потоков изменять одну и ту же область памяти.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Различают две разновидности многозадачности: на основе процессов и на основе потоков. В связи с этим важно понимать отличия между ними.

Процесс отвечает за управление ресурсами, к числу которых относится виртуальная память и дескрипторы Windows, и содержит как минимум один поток. Наличие хотя бы одного потока является обязательным для выполнения любой программы. Поэтому многозадачность на основе процессов — это средство, благодаря которому на компьютере могут параллельно выполняться две программы и более.

Так, многозадачность на основе процессов позволяет одновременно выполнять программы текстового редактора, электронных таблиц и просмотра содержимого в Интернете. При организации многозадачности на основе процессов программа является наименьшей единицей кода, выполнение которой может координировать планировщик задач.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Поток представляет собой координируемую единицу исполняемого кода. Своим происхождением этот термин обязан понятию "поток исполнения". При организации многозадачности на основе потоков у каждого процесса должен быть по крайней мере один поток, хотя их может быть и больше. Это означает, что в одной программе одновременно могут решаться две задачи и больше. Например, текст может форматироваться в редакторе текста одновременно с его выводом на печать, при условии, что оба эти действия выполняются в двух отдельных потоках.

Отличия в многозадачности на основе процессов и потоков могут быть сведены к следующему: многозадачность на основе процессов организуется для параллельного выполнения программ, а многозадачность на основе потоков — для параллельного выполнения отдельных частей одной программы.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Главное преимущество многопоточной обработки заключается в том, что она позволяет писать программы, которые работают очень эффективно благодаря возможности выгодно использовать время простоя, неизбежно возникающее в ходе выполнения большинства программ. Как известно, большинство устройств ввода-вывода, будь то устройства, подключенные к сетевым портам, накопители на дисках или клавиатура, работают намного медленнее, чем центральный процессор (ЦП). Поэтому большую часть своего времени программе приходится ожидать отправки данных на устройство ввода-вывода или приема информации из него. А благодаря многопоточной обработке программа может решать какую-нибудь другую задачу во время вынужденного простоя.

Например, в то время как одна часть программы отправляет файл через соединение с Интернетом, другая ее часть может выполнять чтение текстовой информации, вводимой с клавиатуры, а третья — осуществлять буферизацию очередного блока отправляемых данных.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Поток может находиться в одном из нескольких состояний. В целом, поток может быть выполняющимся; готовым к выполнению, как только он получит время и ресурсы ЦП; приостановленным, т.е. временно не выполняющимся; возобновленным в дальнейшем; заблокированным в ожидании ресурсов для своего выполнения; а также завершенным, когда его выполнение окончено и не может быть возобновлено.

В среде .NET Framework определены две разновидности потоков: приоритетный и фоновый. По умолчанию создаваемый поток автоматически становится приоритетным, но его можно сделать фоновым. *Единственное отличие приоритетных потоков от фоновых заключается в том, что фоновый поток автоматически завершается, если в его процессе остановлены все приоритетные потоки*.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

В связи с организацией многозадачности на основе потоков возникает потребность в особого рода режиме, который называется синхронизацией и позволяет координировать выполнение потоков вполне определенным образом. Для такой синхронизации в C# предусмотрена отдельная подсистема.

Все процессы состоят хотя бы из одного потока, который обычно называют основным, поскольку именно с него начинается выполнение программы. Из основного потока можно создать другие потоки.

В языке C# и среде .NET Framework поддерживаются обе разновидности многозадачности: на основе процессов и на основе потоков. Поэтому средствами C# можно создавать как процессы, так и потоки, а также управлять и теми и другими.

Намного более важной оказывается поддержка в C# многопоточной обработки, благодаря которой упрощается написание высокопроизводительных, многопоточных программ на C# по сравнению с некоторыми другими языками программирования.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен `System.Threading`. В нем определен класс, представляющий отдельный поток - класс `Thread`.

Класс `Thread` определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Основные свойства класса Thread:

Статическое свойство `CurrentContext` позволяет получить контекст, в котором выполняется поток

Статическое свойство `CurrentThread` возвращает ссылку на выполняемый поток

Свойство `IsAlive` указывает, работает ли поток в текущий момент

Свойство `IsBackground` указывает, является ли поток фоновым

Свойство `Name` содержит имя потока

Свойство `Priority` хранит приоритет потока - значение перечисления `ThreadPriority`

Свойство `ThreadState` возвращает состояние потока - одно из значений перечисления `ThreadState`

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Некоторые методы класса Thread:

Статический метод `GetDomain` возвращает ссылку на домен приложения

Статический метод `GetDomainId` возвращает `id` домена приложения, в котором выполняется текущий поток

Статический метод `Sleep` останавливает поток на определенное количество миллисекунд

Метод `Abort` уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки завершенности потока следует опрашивать его свойство `ThreadState`

Метод `Interrupt` прерывает поток на некоторое время

Метод `Join` блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод

Метод `Resume` возобновляет работу ранее приостановленного потока

Метод `Start` запускает поток

Метод `Suspend` приостанавливает поток

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Получение информации о потоке

Используем вышеописанные свойства и методы для получения информации о потоке:

```
using System.Threading;
.....................
static void Main(string[] args)
{
    // получаем текущий поток
    Thread t = Thread.CurrentThread;

    //получаем имя потока
    Console.WriteLine("Имя потока: {0}", t.Name);
    t.Name = "Метод Main";
    Console.WriteLine("Имя потока: {0}", t.Name);

    Console.WriteLine("Запущен ли поток: {0}", t.IsAlive);
    Console.WriteLine("Приоритет потока: {0}", t.Priority);
    Console.WriteLine("Статус потока: {0}", t.ThreadState);

    // получаем домен приложения
    Console.WriteLine("Домен приложения: {0}", Thread.GetDomain().FriendlyName);

    Console.ReadLine();
}
```

В этом случае мы получим примерно следующий вывод:

```
Имя потока:
Имя потока: Метод Main
Запущен ли поток: True
Приоритет потока: Normal
Статус потока: Running
Домен приложения: ThreadApp.vshost.exe
```

Так как по умолчанию свойство `Name` у объектов `Thread` не установлено, то в первом случае мы получаем в качестве значения этого свойства пустую строку.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Статус потока

Статусы потока содержатся в перечислении `ThreadState`:

`Aborted`: поток остановлен, но пока еще окончательно не завершен

`AbortRequested`: для потока вызван метод `Abort`, но остановка потока еще не произошла

`Background`: поток выполняется в фоновом режиме

`Running`: поток запущен и работает (не приостановлен)

`Stopped`: поток завершен

`StopRequested`: поток получил запрос на остановку

`Suspended`: поток приостановлен

`SuspendRequested`: поток получил запрос на приостановку

`Unstarted`: поток еще не был запущен

`WaitSleepJoin`: поток заблокирован в результате действия методов `Sleep` или `Join`

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin. А применяя метод Abort, мы тем самым переведем поток в состояние AbortRequested, а затем Aborted, после чего поток окончательно завершится.

ВВЕДЕНИЕ В МНОГОПОТОЧНОСТЬ. КЛАСС THREAD

Приоритеты потоков

Приоритеты потоков располагаются в перечислении `ThreadPriority`:

`Lowest`

`BelowNormal`

`Normal`

`AboveNormal`

`Highest`

По умолчанию потоку задается значение `Normal`. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет `Highest`. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

СОЗДАНИЕ ПОТОКОВ. ДЕЛЕГАТ THREADSTART

Используя класс `Thread`, мы можем выделить в приложении несколько потоков, которые будут выполняться одновременно.

Во-первых, для запуска нового потока нам надо определить задачу в приложении, которую будет выполнять данный поток. Для этого мы можем добавить новый метод, производящий какие-либо действия.

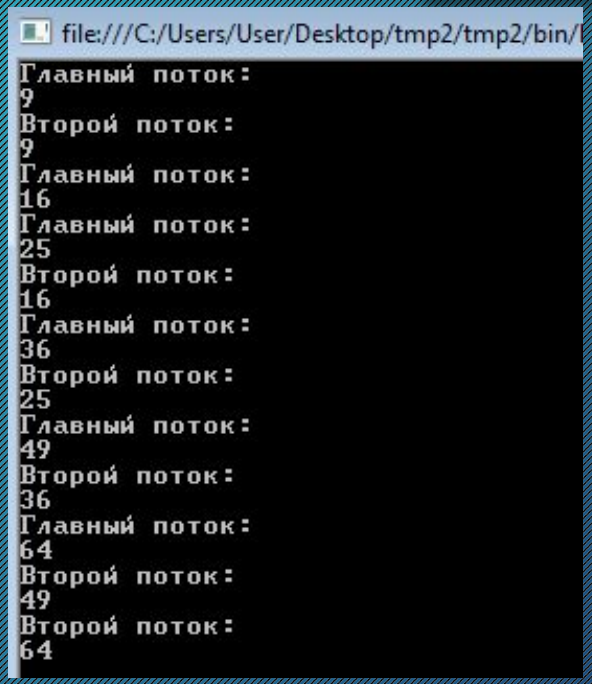
Для создания нового потока используется делегат `ThreadStart`, который получает в качестве параметра метод, который мы определил выше.

И чтобы запустить поток, вызывается метод `Start`. Рассмотрим на примере:

```
ссылок 0
static void Main(string[] args)
{
    // создаем новый поток
    // новый поток будет производить действия, определенные в методе Count
    Thread myThread = new Thread(new ThreadStart(Count));
    myThread.Start(); // запускаем поток

    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("Главный поток:");
        Console.WriteLine(i * i);
        // после каждого умножения с помощью метода Thread.Sleep
        // мы усыпляем поток на 300 миллисекунд
        Thread.Sleep(300);
    }

    Console.ReadLine();
}

ссылка 1
public static void Count()
{
    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("Второй поток:");
        Console.WriteLine(i * i);
        Thread.Sleep(400);
    }
}
```

Здесь новый поток будет производить действия, определенные в методе Count. Чтобы запустить этот метод в качестве второго потока, мы сначала создаем объект потока: `Thread myThread = new Thread(new ThreadStart(Count));`. В конструктор передается делегат `ThreadStart`, который в качестве параметра принимает метод `Count`. И следующей строкой `myThread.Start()` мы запускаем поток. После этого управление передается главному потоку, и выполняются все остальные действия, определенные в методе `Main`.

```
ссылок 0
static void Main(string[] args)
{
    // создаем новый поток
    // новый поток будет производить действия, определенные в методе Count
    Thread myThread = new Thread(new ThreadStart(Count));
    myThread.Start(); // запускаем поток

    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("Главный поток:");
        Console.WriteLine(i * i);
        // после каждого умножения с помощью метода Thread.Sleep
        // мы усыпляем поток на 300 миллисекунд
        Thread.Sleep(300);
    }

    Console.ReadLine();
}

ссылка 1
public static void Count()
{
    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("Второй поток:");
        Console.WriteLine(i * i);
        Thread.Sleep(400);
    }
}
```



Таким образом, в нашей программе будут работать одновременно главный поток, представленный методом Main, и второй поток. Кроме действий по созданию второго потока, в главном потоке также производятся некоторые вычисления. Как только все потоки отработают, программа завершит свое выполнение.

Подобным образом мы можем создать и три, и четыре, и целый набор новых потоков, которые смогут решать те или иные задачи.

СОЗДАНИЕ ПОТОКОВ. ДЕЛЕГАТ THREADSTART

Существует еще одна форма создания потока:

```
Thread myThread = new Thread(Count);
```

Хотя в данном случае явным образом мы не используем делегат `ThreadStart`, но неявно он создается. Компилятор C# выводит делегат из сигнатуры метода `Count` и вызывает соответствующий конструктор.

ПОТОКИ С ПАРАМЕТРАМИ И PARAMETERIZEDTHREADSTART

В предыдущем примере мы рассмотрели, как запускать в отдельных потоках методы без параметров. А что, если нам надо передать какие-нибудь параметры в поток?

Для этой цели используется делегат `ParameterizedThreadStart`. Его действие похоже на функциональность делегата `ThreadStart`. Рассмотрим на примере:

```
ссылок 0
static void Main(string[] args)
{
    int number = 4;
    // создаем новый поток
    Thread myThread = new Thread(new ParameterizedThreadStart(Count));
    // После создания потока мы передаем метод myThread.Start(number);
    // переменную, значение которой хотим передать в поток.
    myThread.Start(number);

    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("Главный поток:");
        Console.WriteLine(i * i);
        Thread.Sleep(300);
    }

    Console.ReadLine();
}

ссылка 1
public static void Count(object x)
{
    for (int i = 1; i < 9; i++)
    {
        int n = (int)x;

        Console.WriteLine("Второй поток:");
        Console.WriteLine(i * n);
        Thread.Sleep(400);
    }
}
```

После создания потока мы передаем метод `myThread.Start(number);` переменную, значение которой хотим передать в поток.

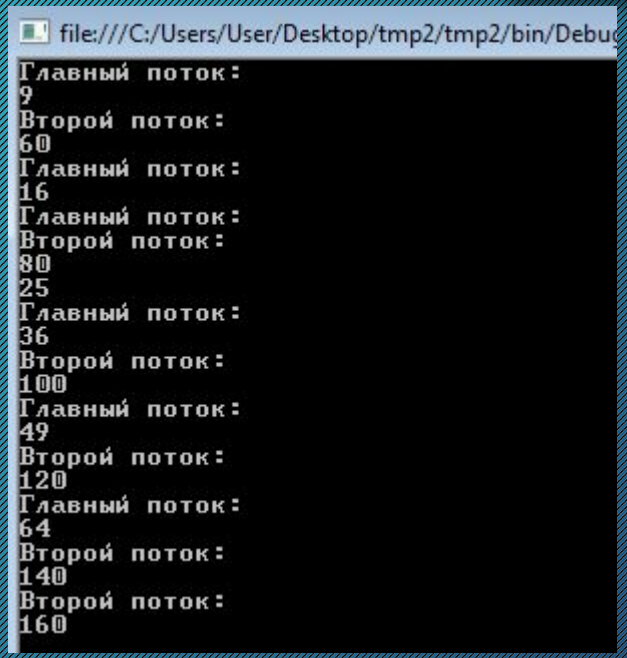
При использовании `ParameterizedThreadStart` мы сталкиваемся с ограничением: мы можем запускать во втором потоке только такой метод, который в качестве единственного параметра принимает объект типа `object`. Поэтому в данном случае нам надо дополнительно привести переданное значение к типу `int`, чтобы его использовать в вычислениях.

ПОТОКИ С ПАРАМЕТРАМИ И PARAMETERIZEDTHREADSTART

Но что делать, если нам надо передать не один, а несколько параметров различного типа? В этом случае на помощь приходит классовый подход:

```
ссылок 0
static void Main(string[] args)
{
    Counter counter = new Counter();
    counter.X = 4; counter.Y = 5;

    Thread myThread = new Thread(new ParameterizedThreadStart(Count));
    myThread.Start(counter);

    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("Главный поток:");
        Console.WriteLine(i * i);
        Thread.Sleep(300);
    }
    Console.ReadLine();
}
ссылка 1
public static void Count(object obj)
{
    for (int i = 1; i < 9; i++)
    {
        Counter c = (Counter)obj;

        Console.WriteLine("Второй поток:");
        Console.WriteLine(i * c.X * c.Y);
        Thread.Sleep(400);
    }
}
```

```
ссылок 4
public class Counter
{
    ссылок 2
    public int X { get; set; }
    ссылок 2
    public int Y { get; set; }
}
```

Сначала определяем специальный класс Counter, объект которого будет передаваться во второй поток, а в методе Main передаем его во второй поток.

Но тут опять же есть одно ограничение: метод Thread.Start не является типобезопасным, то есть мы можем передать в него любой тип, и потом нам придется приводить переданный объект к нужному нам типу.

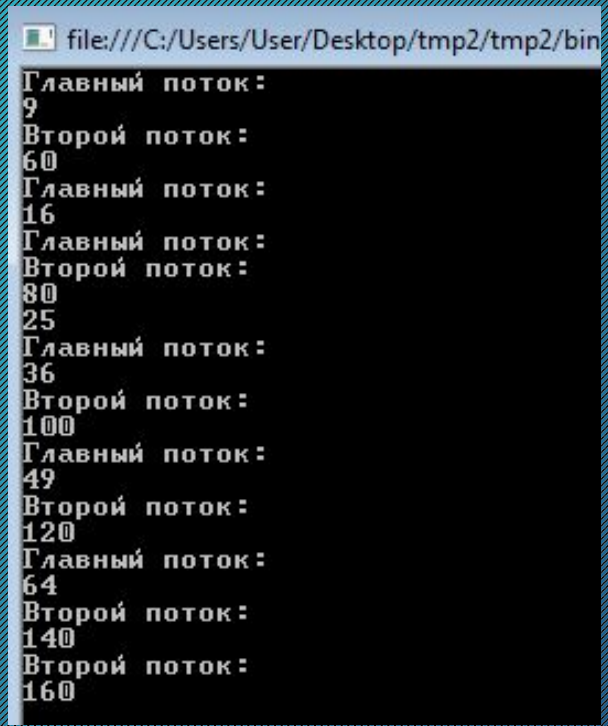
ПОТОКИ С ПАРАМЕТРАМИ И PARAMETERIZEDTHREADSTART

Для решения данной проблемы рекомендуется объявлять все используемые методы и переменные в специальном классе, а в основной программе запускать поток через `ThreadStart`.

Например:

```
class Program
{
    ссылок 0
    static void Main(string[] args)
    {
        Counter counter = new Counter(5, 4);
        Thread myThread = new Thread(new ThreadStart(counter.Count));
        myThread.Start();

        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Главный поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(300);
        }
        Console.ReadLine();
    }
}
```

```
ссылок 3
public class Counter
{
    private int x;
    private int y;
    ссылка 1
    public Counter(int _x, int _y)
    {
        this.x = _x;
        this.y = _y;
    }
    ссылка 1
    public void Count()
    {
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Второй поток:");
            Console.WriteLine(i * x * y);
            Thread.Sleep(400);
        }
    }
}
```

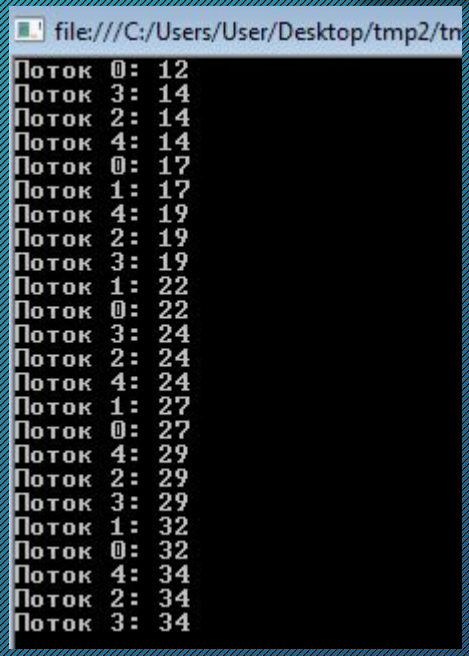


СИНХРОНИЗАЦИЯ ПОТОКОВ

Нередко в потоках используются некоторые разделяемые ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы.

Например:

```
static int x = 0;
ссылок 0
static void Main(string[] args)
{
    // запускаются пять потоков, которые работают с общей переменной x
    for (int i = 0; i < 5; i++)
    {
        Thread myThread = new Thread(Count);
        // задаем имя для потока
        myThread.Name = "Поток " + i.ToString();
        myThread.Start();
    }

    Console.ReadLine();
}
ссылка 1
public static void Count()
{
    x = 1;
    for (int i = 1; i < 9; i++)
    {
        // Thread.CurrentThread.Name - возвращает имя текущего потока
        Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
        x++;
        Thread.Sleep(100);
    }
}
```

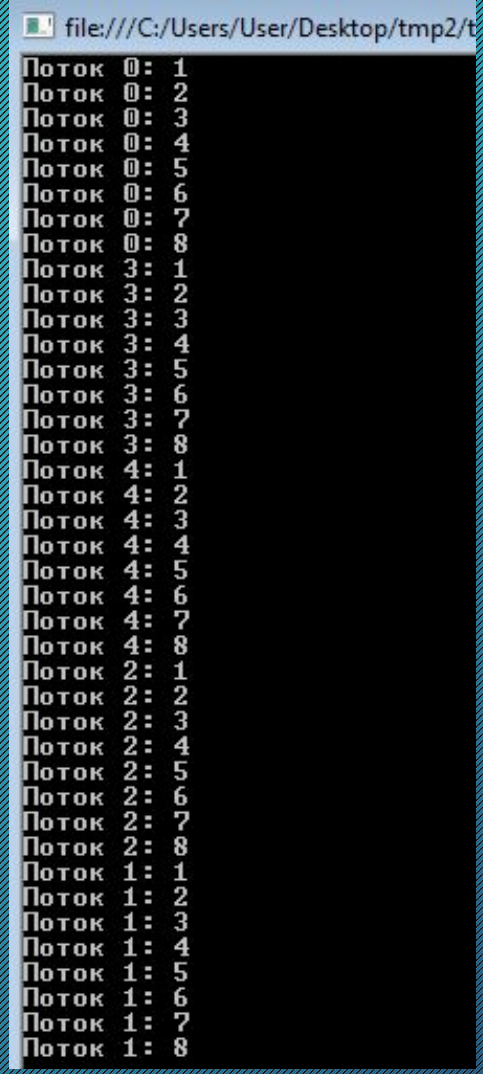



Здесь у нас запускаются пять потоков, которые работают с общей переменной x. И мы предполагаем, что метод выведет все значения x от 1 до 8. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым.

СИНХРОНИЗАЦИЯ ПОТОКОВ

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово `lock`. Оператор `lock` определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. И мы можем переделать предыдущий пример следующим образом:

```
static int x = 0;
// объект-заглушка для блокировки
static object locker = new object();
ссылок 0
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Thread myThread = new Thread(Count);
        // задаем имя потока
        myThread.Name = "Поток " + i.ToString();
        myThread.Start();
    }

    Console.ReadLine();
}
ссылка 1
public static void Count()
{
    lock (locker)
    {
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            // Thread.CurrentThread.Name - возвращает имя текущего потока
            Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
            x++;
            Thread.Sleep(100);
        }
    }
}
```



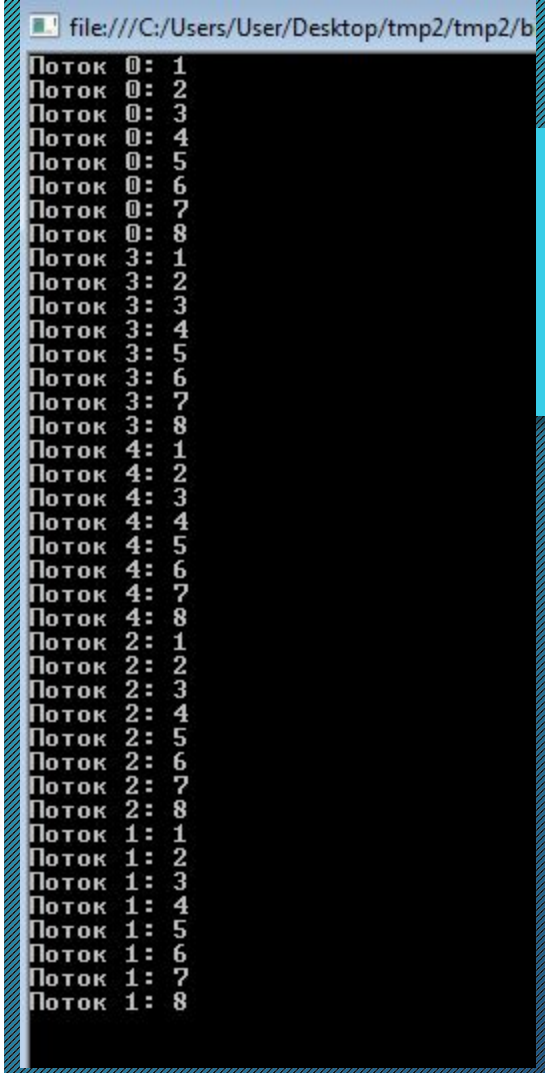
Для блокировки с ключевым словом `lock` используется объект-заглушка, в данном случае это переменная `locker`. Когда выполнение доходит до оператора `lock`, объект `locker` блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток. После окончания работы блока кода, объект `locker` освобождается и становится доступным для других потоков.

МОНИТОРЫ

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом System.Threading.Monitor.

Фактически конструкция оператора lock из прошлого примера инкапсулирует в себе синтаксис использования мониторов. А рассмотренный пример будет эквивалентен следующему коду:

```
static int x = 0;
static object locker = new object();
ссылок 0
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Thread myThread = new Thread(Count);
        myThread.Name = "Поток " + i.ToString();
        myThread.Start();
    }
    Console.ReadLine();
}
ссылка 1
public static void Count()
{
    try
    {
        Monitor.Enter(locker);
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
            x++;
            Thread.Sleep(100);
        }
    }
    finally
    {
        Monitor.Exit(locker);
    }
}
```



Метод `Monitor.Enter` блокирует объект locker так же, как это делает оператор `lock`. А в блоке `try...finally` с помощью метода `Monitor.Exit` происходит освобождение объекта `locker`, и он становится доступным для других потоков.

МОНИТОРЫ

Кроме блокировки и разблокировки объекта класс Monitor имеет еще ряд методов, которые позволяют управлять синхронизацией потоков. Так, метод Monitor.Wait освобождает блокировку объекта и переводит поток в очередь ожидания объекта. Следующий поток в очереди готовности объекта блокирует данный объект. А все потоки, которые вызвали метод Wait, остаются в очереди ожидания, пока не получат сигнала от метода Monitor.Pulse или Monitor.PulseAll, посланного владельцем блокировки. Если метод Monitor.Pulse отправлен, поток, находящийся во главе очереди ожидания, получает сигнал и блокирует освободившийся объект. Если же метод Monitor.PulseAll отправлен, то все потоки, находящиеся в очереди ожидания, получают сигнал и переходят в очередь готовности, где им снова разрешается получать блокировку объекта.

КЛАСС AUTORESETEVENT

Класс AutoResetEvent также служит целям синхронизации потоков. Этот класс является оберткой над объектом ОС Windows "событие" и позволяет переключить данный объект-событие из сигнального в несигнальное состояние. Так, предыдущий пример мы можем переписать с использованием AutoResetEvent следующим образом:

```
// создаем объект типа AutoResetEvent
// передавая в конструктор значение true, мы тем самым указываем,
// что создаваемый объект изначально будет в сигнальном состоянии
static AutoResetEvent waitHandler = new AutoResetEvent(true);
static int x = 0;
ссылок 0
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Thread myThread = new Thread(Count);
        myThread.Name = "Поток " + i.ToString();
        myThread.Start();
    }
    Console.ReadLine();
}
ссылка 1
public static void Count()
{
    // метод WaitOne указывает, что текущий поток переводится в состояние ожидания,
    // пока объект waitHandler не будет переведен в сигнальное состояние
    // и так все потоки у нас переводятся в состояние ожидания
    waitHandler.WaitOne();
    x = 1;
    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
        x++;
        Thread.Sleep(100);
    }
    // метод waitHandler.Set, который уведомляет все ожидающие потоки,
    // что объект waitHandler снова находится в сигнальном состоянии,
    // и один из потоков "захватывает" данный объект, переводит в несигнальное состояние и выполняет свой код.
    waitHandler.Set();
}
```

Во-первых, создаем переменную типа `AutoResetEvent`. Передавая в конструктор значение `true`, мы тем самым указываем, что создаваемый объект изначально будет в сигнальном состоянии. Когда начинает работать поток, то первым делом срабатывает определенный в методе `Count` вызов `waitHandler.WaitOne()`. Метод `WaitOne` указывает, что текущий поток переводится в состояние ожидания, пока объект `waitHandler` не будет переведен в сигнальное состояние. И так все потоки у нас переводятся в состояние ожидания.

```
// создаем объект типа AutoResetEvent
// передавая в конструктор значение true, мы тем самым указываем,
// что создаваемый объект изначально будет в сигнальном состоянии
static AutoResetEvent waitHandler = new AutoResetEvent(true);
static int x = 0;
ссылок 0
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Thread myThread = new Thread(Count);
        myThread.Name = "Поток " + i.ToString();
        myThread.Start();
    }
    Console.ReadLine();
}
ссылка 1
public static void Count()
{
    // метод WaitOne указывает, что текущий поток переводится в состояние ожидания,
    // пока объект waitHandler не будет переведен в сигнальное состояние
    // и так все потоки у нас переводятся в состояние ожидания
    waitHandler.WaitOne();
    x = 1;
    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
        x++;
        Thread.Sleep(100);
    }
    // метод waitHandler.Set, который уведомляет все ожидающие потоки,
    // что объект waitHandler снова находится в сигнальном состоянии,
    // и один из потоков "захватывает" данный объект, переводит в несигнальное состояние и выполняет свой код.
    waitHandler.Set();
}
```

```
file:///C:/Users/User/Desktop/tm
Поток 0: 1
Поток 0: 2
Поток 0: 3
Поток 0: 4
Поток 0: 5
Поток 0: 6
Поток 0: 7
Поток 0: 8
Поток 4: 1
Поток 4: 2
Поток 4: 3
Поток 4: 4
Поток 4: 5
Поток 4: 6
Поток 4: 7
Поток 4: 8
Поток 3: 1
Поток 3: 2
Поток 3: 3
Поток 3: 4
Поток 3: 5
Поток 3: 6
Поток 3: 7
Поток 3: 8
Поток 2: 1
Поток 2: 2
Поток 2: 3
Поток 2: 4
Поток 2: 5
Поток 2: 6
Поток 2: 7
Поток 2: 8
Поток 1: 1
Поток 1: 2
Поток 1: 3
Поток 1: 4
Поток 1: 5
Поток 1: 6
Поток 1: 7
Поток 1: 8
```

После завершения работы вызывается метод `waitHandler.Set`, который уведомляет все ожидающие потоки, что объект `waitHandler` снова находится в сигнальном состоянии, и один из потоков "захватывает" данный объект, переводит в несигнальное состояние и выполняет свой код. А остальные потоки снова ожидают. Так как в конструкторе `AutoResetEvent` мы указываем, что объект изначально находится в сигнальном состоянии, то первый из очереди потоков захватывает данный объект и начинает выполнять свой код.

КЛАСС AUTORESETEVENT

Но если бы мы написали

```
AutoResetEvent waitHandler = new AutoResetEvent(false),
```

тогда объект изначально был бы в несигнальном состоянии, а поскольку все потоки блокируются методом waitHandler.WaitOne() до ожидания сигнала, то у нас попросту случилась бы блокировка программы, и программа не выполняла бы никаких действий.

Если у нас в программе используются несколько объектов AutoResetEvent, то мы можем использовать для отслеживания состояния этих объектов методы WaitAll и WaitAny, которые в качестве параметра принимают массив объектов класса WaitHandle - базового класса для AutoResetEvent.

Так, мы тоже можем использовать WaitAll в вышеприведенном примере. Для этого надо строку waitHandler.WaitOne(); заменить на следующую:

```
AutoResetEvent.WaitAll(new WaitHandle[] {waitHandler});
```

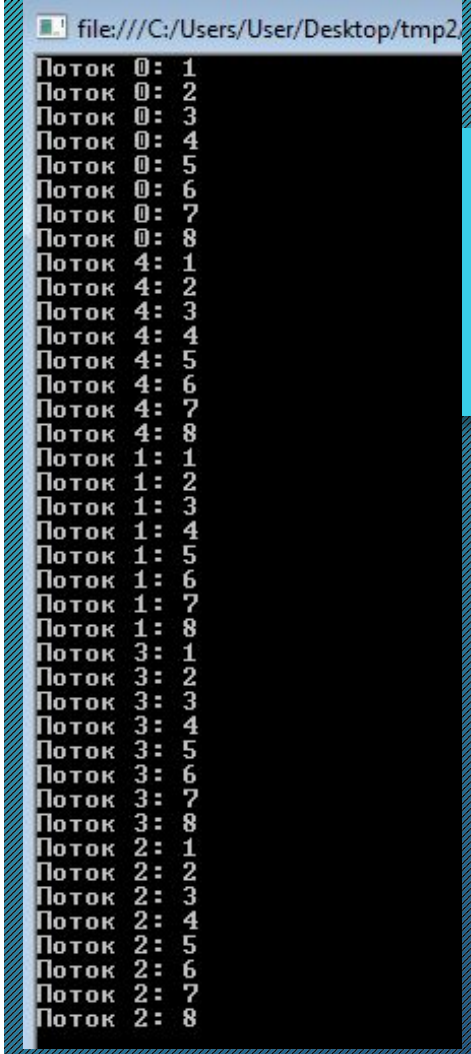
МЬЮТЕКСЫ

Еще один инструмент управления синхронизацией потоков представляет класс `Mutex`, также находящийся в пространстве имен `System.Threading`. Данный класс является классом-оболочкой над соответствующим объектом ОС Windows "мьютекс". Перепишем предыдущий пример, используя мьютексы:

```
// создаем объект мьютекса
static Mutex mutexObj = new Mutex();
static int x = 0;

ссылок 0
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Thread myThread = new Thread(Count);
        myThread.Name = "Поток " + i.ToString();
        myThread.Start();
    }

    Console.ReadLine();
}
ссылка 1
public static void Count()
{
    // Метод mutexObj.WaitOne() приостанавливает выполнение потока до тех пор,
    // пока не будет получен мьютекс mutexObj
    mutexObj.WaitOne();
    x = 1;
    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
        x++;
        Thread.Sleep(100);
    }
    // освобождение мьютекса
    mutexObj.ReleaseMutex();
}
```


Сначала создаем объект мьютекса: `Mutex mutexObj = new Mutex()`.
Основную работу по синхронизации выполняют методы `WaitOne()` и `ReleaseMutex()`. Метод `mutexObj.WaitOne()` приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс `mutexObj`.
После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода `mutexObj.ReleaseMutex()`

```
// создаем объект мьютекса
static Mutex mutexObj = new Mutex();
static int x = 0;

ссылок 0
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Thread myThread = new Thread(Count);
        myThread.Name = "Поток " + i.ToString();
        myThread.Start();
    }

    Console.ReadLine();
}
ссылка 1
public static void Count()
{
    // Метод mutexObj.WaitOne() приостанавливает выполнение потока до тех пор,
    // пока не будет получен мьютекс mutexObj
    mutexObj.WaitOne();
    x = 1;
    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
        x++;
        Thread.Sleep(100);
    }
    // освобождение мьютекса
    mutexObj.ReleaseMutex();
}
```



Таким образом, когда выполнение дойдет до вызова `mutexObj.WaitOne()`, поток будет ожидать, пока не освободится мьютекс. И после его получения продолжит выполнять свою работу.

МЬЮТЕКСЫ

Мы использовали мьютекс для синхронизации потоков. Однако замечательная черта мьютексов состоит также в том, что они могут также применяться не только внутри одного процесса, но и между процессами. Типичный пример - создание приложения, которое можно запустить только один раз. Создадим подобное приложение:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
using System.Runtime.InteropServices;
using System.Reflection;

namespace example
{
    ссылок 0
    class Program
    {
        ссылок 0
        static void Main(string[] args)
        {
            bool existed;
            // получаем GIUD приложения
            string guid = Marshal.GetTypeLibGuidForAssembly(Assembly.GetExecutingAssembly()).ToString();

            Mutex mutexObj = new Mutex(true, guid, out existed);

            if (existed)
            {
                Console.WriteLine("Приложение работает");
            }
            else
            {
                Console.WriteLine("Приложение уже было запущено. И сейчас оно будет закрыто.");
                Thread.Sleep(3000);
                return;
            }
            Console.ReadLine();
        }
```

В данном случае для создания мьютекса мы используем другую перегрузку конструктора. Значение true, которое передается в качестве первого параметра конструктора, указывает, что приложение будет запрашивать владение мьютексом. Второй параметр указывает на уникальное имя мьютекса. В данном случае в качестве имени выбран guid приложения, то есть глобальный уникальный идентификатор.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
using System.Runtime.InteropServices;
using System.Reflection;

namespace example
{
    ссылок 0
    class Program
    {
        ссылок 0
        static void Main(string[] args)
        {
            bool existed;
            // получаем GIUD приложения
            string guid = Marshal.GetTypeLibGuidForAssembly(Assembly.GetExecutingAssembly()).ToString();

            Mutex mutexObj = new Mutex(true, guid, out existed);

            if (existed)
            {
                Console.WriteLine("Приложение работает");
            }
            else
            {
                Console.WriteLine("Приложение уже было запущено. И сейчас оно будет закрыто.");
                Thread.Sleep(3000);
                return;
            }
            Console.ReadLine();
        }
```

Третий параметр возвращает значение из конструктора. Если он равен true, то это означает, что мьютекс запрошен и получен. А если false - то запрос на владение мьютексом отклонен. И после создания мьютекса, если мы запустим вторую копию приложения, то она будет закрыта. И в один момент времени сможет работать только одна копия программы.

СЕМАФОРЫ

Еще один инструмент, который предлагает нам платформа .NET для управления синхронизацией, представляют семафоры. Семафоры позволяют ограничить доступ определенным количеством объектов.

Например, у нас такая задача: есть некоторое число читателей, которые приходят в библиотеку три раза в день и что-то там читают. И пусть у нас будет ограничение, что единовременно в библиотеке не может находиться больше трех читателей. Данную задачу очень легко решить с помощью семафоров:

```
ссылок 3
class Reader
{
    static Semaphore sem = new Semaphore(3, 3); // создаем семафор
    Thread myThread;
    int count = 3;// счетчик чтения
    ссылка 1
    public Reader(int i)
    {
        myThread = new Thread(Read);
        myThread.Name = "Читатель " + i.ToString();
        myThread.Start();
    }
    ссылка 1
    public void Read()
    {
        while (count > 0)
        {
            sem.WaitOne();
            Console.WriteLine("{0} входит в библиотеку", Thread.CurrentThread.Name);
            Console.WriteLine("{0} читает", Thread.CurrentThread.Name);
            Thread.Sleep(1000);
            Console.WriteLine("{0} покидает библиотеку", Thread.CurrentThread.Name);
            sem.Release();
            count--;
            Thread.Sleep(1000);
        }
    }
}
```

```
ссылок 0
class Program
{
    ссылок 0
    static void Main(string[] args)
    {
        for (int i = 1; i < 6; i++)
        {
            Reader reader = new Reader(i);
        }
        Console.ReadLine();
    }
}
```

В данной программе читатель представлен классом `Reader`. Он инкапсулирует всю функциональность, связанную с потоками, через переменную `Thread myThread`. Для создания семафора используется класс `Semaphore`: `static Semaphore sem = new Semaphore(3, 3);`. Его конструктор принимает два параметра: первый указывает, какому числу объектов изначально будет доступен семафор, а второй параметр указывает, какой максимальное число объектов будет использовать данный семафор. В данном случае у нас только три читателя могут одновременно находиться в библиотеке, поэтому максимальное число равно 3.

```
ссылок 3
class Reader
{
    static Semaphore sem = new Semaphore(3, 3); // создаем семафор
    Thread myThread;
    int count = 3;// счетчик чтения
    ссылка 1
    public Reader(int i)
    {
        myThread = new Thread(Read);
        myThread.Name = "Читатель " + i.ToString();
        myThread.Start();
    }
    ссылка 1
    public void Read()
    {
        while (count > 0)
        {
            sem.WaitOne();
            Console.WriteLine("{0} входит в библиотеку", Thread.CurrentThread.Name);
            Console.WriteLine("{0} читает", Thread.CurrentThread.Name);
            Thread.Sleep(1000);
            Console.WriteLine("{0} покидает библиотеку", Thread.CurrentThread.Name);
            sem.Release();
            count--;
            Thread.Sleep(1000);
        }
    }
}
```

```
ссылок 0
class Program
{
    ссылок 0
    static void Main(string[] args)
    {
        for (int i = 1; i < 6; i++)
        {
            Reader reader = new Reader(i);
        }
        Console.ReadLine();
    }
}
```

Основной функционал сосредоточен в методе `Read`, который и выполняется в потоке. В начале для ожидания получения семафора используется метод `sem.WaitOne()`. После того, как в семафоре освободится место, данный поток заполняет свободное место и начинает выполнять все дальнейшие действия. После окончания чтения мы высвобождаем семафор с помощью метода `sem.Release()`. После этого в семафоре освобождается одно место, которое заполняет другой поток.

А в методе `Main` нам остается только создать читателей, которые запускают соответствующие потоки.

```
ссылок 3
class Reader
{
    static Semaphore sem = new Semaphore(3, 3); // создаем семафор
    Thread myThread;
    int count = 3;// счетчик чтения
    ссылка 1
    public Reader(int i)
    {
        myThread = new Thread(Read);
        myThread.Name = "Читатель " + i.ToString();
        myThread.Start();
    }
    ссылка 1
    public void Read()
    {
        while (count > 0)
        {
            sem.WaitOne();
            Console.WriteLine("{0} входит в библиотеку", Thread.CurrentThread.Name);
            Console.WriteLine("{0} читает", Thread.CurrentThread.Name);
            Thread.Sleep(1000);
            Console.WriteLine("{0} покидает библиотеку", Thread.CurrentThread.Name);
            sem.Release();
            count--;
            Thread.Sleep(1000);
        }
    }
}
```

```
ссылок 0
class Program
{
    ссылок 0
    static void Main(string[] args)
    {
        for (int i = 1; i < 6; i++)
        {
            Reader reader = new Reader(i);
        }
        Console.ReadLine();
    }
}
```

file:///C:/Users/User/Desktop/tmp2/tmp2/bin/Debug/tmp2.EXE

```
Читатель 4 читает
Читатель 1 входит в библиотеку
Читатель 1 читает
Читатель 5 покидает библиотеку
Читатель 3 входит в библиотеку
Читатель 3 читает
Читатель 4 покидает библиотеку
Читатель 2 входит в библиотеку
Читатель 2 читает
Читатель 1 покидает библиотеку
Читатель 5 входит в библиотеку
Читатель 5 читает
Читатель 3 покидает библиотеку
Читатель 2 покидает библиотеку
Читатель 4 входит в библиотеку
Читатель 4 читает
Читатель 1 входит в библиотеку
Читатель 1 читает
Читатель 5 покидает библиотеку
Читатель 3 входит в библиотеку
Читатель 3 читает
Читатель 4 покидает библиотеку
Читатель 2 входит в библиотеку
Читатель 2 читает
Читатель 1 покидает библиотеку
Читатель 5 входит в библиотеку
Читатель 5 читает
Читатель 3 покидает библиотеку
Читатель 2 покидает библиотеку
Читатель 4 входит в библиотеку
Читатель 4 читает
Читатель 5 покидает библиотеку
Читатель 4 покидает библиотеку
```

ИСПОЛЬЗОВАНИЕ ТАЙМЕРОВ

Одним из важнейших классов, находящихся в пространстве имени System.Threading, является класс Timer. Данный класс позволяет запускать определенные действия по истечению некоторого периода времени.

Например, нам надо запускать какой-нибудь метод через каждые 2000 миллисекунд, то есть раз в две секунды:

```
ссылок 0
static void Main(string[] args)
{
    int num = 0;
    // устанавливаем метод обратного вызова
    // создается объект делегата TimerCallback,
    // который в качестве параметра принимает метод
    TimerCallback tm = new TimerCallback(Count);
    // создаем таймер
    Timer timer = new Timer(tm, num, 0, 2000);

    Console.ReadLine();
}
ссылка 1
public static void Count(object obj)
{
    int x = (int)obj;
    for (int i = 1; i < 9; i++, x++)
    {
        Console.WriteLine("{0}", x * i);
    }
}
```

Первым делом создается объект делегата `TimerCallback`, который в качестве параметра принимает метод. Причем данный метод должен в качестве параметра принимать объект типа `object`. И затем создается таймер. Данная перегрузка конструктора таймера принимает четыре параметра: объект делегата `TimerCallback`; объект, передаваемый в качестве параметра в метод `Count`; количество миллисекунд, через которое таймер будет запускаться (в данном случае таймер будет запускать немедленно после создания, так как в качестве значения используется 0); интервал между вызовами метода `Count`. И, таким образом, после запуска программы каждые две секунды будет срабатывать метод `Count`.

ИСПОЛЬЗОВАНИЕ ТАЙМЕРОВ

Если бы нам не надо было бы использовать параметр obj у метода Count, то при создании таймера мы могли бы указывать в качестве соответствующего параметра значение null:

```
Timer timer = new Timer(tm, null, 0, 2000);
```