**Многопоточность**

**Многопоточность** – это разделение потоков, которые смогут выполнять разные действия параллельно друг другу в одно и тоже время.

**Поток** (thread) – это независимая 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.

Классы, поддерживающие многопоточное программирование, определены в пространстве имен **System.Threading**. Поэтому любая многопоточная программа на C# включает в себя следующую строку кода:

using 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. Основные **свойства класса**:

* 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**

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

**Получение информации о потоке**

Используем вышеописанные свойства и методы для получения информации о потоке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | 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.

**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

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.

**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.

Программа на C# запускается как единственный поток, автоматически создаваемый CLR и операционной системой (“главный” поток), и становится многопоточной при помощи создания дополнительных потоков. Вот простой пример и его вывод

static void Main(string[] args)

{

// получаем текущий поток

Thread t = new Thread(WriteY);

t.Start(); // Выполнить WriteY в новом потоке

int i=0;

while (i < 1000)

{

Console.Write("x"); // Все время печатать 'x'

i++;

}

Console.ReadKey();

}

static void WriteY()

{

int i=0;

while (i < 1000)

{

Console.Write("y"); // Все время печатать 'y'

i++;

}

}

Вывод: xxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxyyy yyyyyyyyyyyyyxxxxyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxx xxxxxxxxxyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyx xxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxyyyy yyyyyyyyyyyyxxxxy...

В главном потоке создается новый поток t, исполняющий метод, который непрерывно печатает символ ‘y’. Одновременно главный поток непрерывно печатает символ ‘x’.

Временная приостановка (блокирование) – основной способ координации, или синхронизации действий потоков. Ожидание эксклюзивной блокировки – это одна из причин, по которым поток может блокироваться. Другая причина – если поток приостанавливается (Sleep) на заданный промежуток времени:

Thread.Sleep(TimeSpan.FromSeconds(10)); // Блокировка на 10сек

Также поток может ожидать завершения другого потока, вызывая его метод Join:

t.Join(); // Ожидаем завершения потока

Будучи блокированным, поток не потребляет ресурсов CPU.

Управление многопоточностью осуществляет **планировщик потоков**, эту функцию CLR обычно делегирует операционной системе. Планировщик потоков гарантирует, что активным потокам выделяется соответствующее время на выполнение, а потоки, ожидающие или блокированные, к примеру, на ожидании эксклюзивной блокировки, или пользовательского ввода – не потребляют времени CPU.

На однопроцессорных компьютерах планировщик потоков использует квантование времени – быстрое переключение между выполнением каждого из активных потоков. Это приводит к непредсказуемому поведению, как в самом первом примере, где каждая последовательность символов „X‟ и „Y‟ соответствует кванту времени, выделенному потоку. В Windows XP типичное значение кванта времени – десятки миллисекунд – выбрано как намного большее, чем затраты CPU на переключение контекста между потоками (несколько микросекунд).

Говорят, что поток вытесняется, когда его выполнение приостанавливается из-за внешних факторов типа квантования времени. В большинстве случаев поток не может контролировать, когда и где он будет вытеснен.

**Потоки vs. процессы**

Все потоки одного приложения логически содержатся в пределах процесса – модуля операционной системы, в котором исполняется приложение. В некоторых аспектах потоки и процессы схожи – например, время разделяется между процессами, исполняющимися на одном компьютере, так же, как между потоками одного C#-приложения. Ключевое различие состоит в том, что процессы полностью изолированы друг от друга. Потоки разделяют память (кучу) с другими потоками этого же приложения. Благодаря этому один поток может поставлять данные в фоновом режиме, а другой – показывать эти данные по мере их поступления.

**Создание и запуск потоков**

Для создания потоков используется конструктор класса Thread, принимающий в качестве параметра делегат типа ThreadStart, указывающий метод, который нужно выполнить. Делегат **ThreadStart** определяется так: public delegate void ThreadStart();

Вызов метода Start начинает выполнение потока. Поток продолжается до выхода из исполняемого метода. Вот пример, использующий полный синтаксис C# для создания делегата ThreadStart:

Thread t = new Thread(new ThreadStart(WriteY));

Поток можно создать, используя для присваивания значений делегатам более удобный сокращенный синтаксис C#:

Thread t = new Thread(WriteY);

Поток имеет свойство IsAlive, возвращающее true после вызова Start() и до завершения потока.

Поток, который закончил исполнение, не может быть начат заново.

В случае, если нужно передать параметры в поток, используется **ParameterizedThreadStart**.

static void Main(string[] args)

    {

        int number = 4;

        // создаем новый поток

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

        myThread.Start(number);

        for (int i = 1; i < 9; i++)

        {

            Console.WriteLine("Главный поток:");

            Console.WriteLine(i \* i);

            Thread.Sleep(300);

        }

        Console.ReadLine();

    }

    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);

        }

    }

**Синхронизация потоков**

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

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.

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.

Итоговый пример:

#region[Пример 1. Создание, именование, остановка потоков]

//class Program

//{

// static void Main(string[] args)

// {

// Thread t = new Thread(WriteY);// получаем текущий поток

// t.Start(); // Выполнить WriteY в новом потоке

// //Thread.Sleep(TimeSpan.FromSeconds(10)); // Блокировка на 10сек

// //t.Join(); // Ожидаем завершения потока

// //Thread.CurrentThread.Name = "Patok1";

// //t.Name = "Patok2";

// int i = 0;

// while (i < 1000)

// {

// Console.Write("x"); // Все время печатать 'x'

// i++;

// }

// //Console.WriteLine("Hello from " + Thread.CurrentThread.Name);

// Console.ReadKey();

// }

// static void WriteY()

// {

// int i = 0;

// while (i < 1000)

// {

// Console.Write("y"); // Все время печатать 'y'

// i++;

// }

// //Console.WriteLine("Hello from " + Thread.CurrentThread.Name);

// }

//}

#endregion

#region[Пример 2. Синхронизация]

//class Program

//{

// static int x = 0;

// static object locker = new object();

// 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();

// }

// public static void Count()

// {

// lock (locker)//код блокируется и становится недоступным для других потоков до завершения работы текущего потока

// {

// x = 1;

// for (int i = 1; i < 9; i++)

// {

// Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

// x++;

// Thread.Sleep(100);

// }

// }

// }

// }

#endregion