**Класс Thread**

Система многопоточной обработки основывается на классе Thread, который инкапсулирует

поток исполнения. Класс Thread является *герметичным,* т.е. он не может

наследоваться. В классе Thread определен ряд методов и свойств, предназначенных

для управления потоками. На протяжении всей этой главы будут рассмотрены наиболее

часто используемые члены данного класса.

**Создание и запуск потока**

Для создания потока достаточно получить экземпляр объекта типа Thread,

т.е. класса, определенного в пространстве имен System.Threading. Ниже приведена

простейшая форма конструктора класса Thread:

public Thread(ThreadStart запуск)

где *запуск* — это имя метода, вызываемого с целью начать выполнение потока,

a ThreadStart — делегат, определенный в среде .NET Framework, как показано

ниже.

public delegate void ThreadStart()

Следовательно, метод, указываемый в качестве точки входа в поток, должен иметь

возвращаемый тип void и не принимать никаких аргументов.

Вновь созданный новый поток не начнет выполняться до тех пор, пока не будет

вызван его метод Start(), определяемый в классе Thread. Существуют две формы

объявления метода Start(). Ниже приведена одна из них.

public void Start()

Однажды начавшись, поток будет выполняться до тех пор, пока не произойдет

возврат из метода, на который указывает *запуск.* Таким образом, после возврата из

этого метода поток автоматически прекращается. Если же попытаться вызвать метод

Start() для потока, который уже начался, это приведет к генерированию исключения

ThreadStateException.

В приведенном ниже примере программы создается и начинает выполняться новый

поток.

(***glava23\_1***)

using System.Threading;

class MyThread

{

public int Count;

string thrdName;

public MyThread(string name)

{

Count = 0;

thrdName = name;

}

//start thread point

public void Run()

{

Console.WriteLine(thrdName + " has started.");

do

{

Thread.Sleep(500);

Console.WriteLine("In thread " + thrdName + ", Count = " +

Count);

Count++;

} while (Count < 10);

Console.WriteLine(thrdName + " has ended.");

}

}

class MultiThread

{

static void Main()

{

Console.WriteLine("Main thread started.");

//first build object of MyThread

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

//next build thread from this object

Thread newThrd = new Thread(mt.Run);

//and finally start thread

newThrd.Start();

do

{

Console.Write(".");

Thread.Sleep(100);

} while (mt.Count != 10);

Console.WriteLine("Main thread ended.");

}

}

(Короче говоря, два цикла работают одновременно. Сначала запускается цикл который находится в MyThread, и параллельно с ним цикл в Main())

В самом ее начале определяется класс MyThread, предназначенный для создания второго потока исполнения. В методе Run() этого класса организуется цикл для подсчета от 0 до 9. Обратите

внимание на вызов статического метода Sleep(), определенного в классе Thread.

Этот метод обусловливает приостановление того потока, из которого он был вызван,

на определенный период времени, указываемый в миллисекундах. Когда приостанавливается

один поток, может выполняться другой. В данной программе используется

следующая форма метода Sleep():

public static void Sleep(int миллисекунд\_простоя)

где *миллисекунд\_простоя* обозначает период времени, на который приостанавливается

выполнение потока. Если указанное количество *миллисекунд\_простоя* равно

нулю, то вызывающий поток приостанавливается лишь для того, чтобы предоставить

возможность для выполнения потока, ожидающего своей очереди.

В методе Main() новый объект типа Thread создается с помощью приведенной

ниже последовательности операторов.

//first build object of MyThread

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

//next build thread from this object

Thread newThrd = new Thread(mt.Run);

//and finally start thread

newThrd.Start();

Как следует из комментариев к приведенному выше фрагменту кода, сначала создается

объект типа MyThread. Затем этот объект используется для создания объекта

типа Thread, для чего конструктору этого объекта в качестве точки входа передается

метод mt.Run(). И наконец, выполнение потока начинается с вызова метода Start().

Благодаря этому метод mt.Run() выполняется в своем собственном потоке. После вызова

метода Start() выполнение основного потока возвращается к методу Main(),

где начинается цикл do-while. Оба потока продолжают выполняться, совместно используя

ЦП, вплоть до окончания цикла. Ниже приведен результат выполнения данной

программы.

Зачастую в многопоточной программе требуется, чтобы основной поток был последним

потоком, завершающим ее выполнение. Формально программа продолжает

выполняться до тех пор, пока не завершатся все ее приоритетные потоки. Поэтому

требовать, чтобы основной поток завершал выполнение программы, совсем не обязательно.

Тем не менее этого правила принято придерживаться в многопоточном

программировании, поскольку оно явно определяет конечную точку программы.

В рассмотренной выше программе предпринята попытка сделать основной поток

завершающим ее выполнение. Для этой цели значение переменной Count проверяется

в цикле do-while внутри метода Main(), и как только это значение оказывается

равным 10, цикл завершается и происходит поочередный возврат из методов Sleep().

Но такой подход далек от совершенства, поэтому далее в этой главе будут представлены

более совершенные способы организации ожидания одного потока до завершения

другого.

**Простые способы усовершенствования многопоточной программы**

Рассмотренная выше программа вполне работоспособна, но ее можно сделать более

эффективной, внеся ряд простых усовершенствований. Во-первых, можно сделать

так, чтобы выполнение потока начиналось сразу же после его создания. Для этого достаточно

получить экземпляр объекта типа Thread в конструкторе класса MyThread.

И во-вторых, в классе MyThread совсем не обязательно хранить имя потока, поскольку

для этой цели в классе Thread специально определено свойство Name.

public string Name { get; set; }

Свойство Name доступно для записи и чтения и поэтому может сложить как для

запоминания, так и для считывания имени потока.

Ниже приведена версия предыдущей программы, в которую внесены упомянутые

выше усовершенствования.

(***glava23\_2***)

using System.Threading;

class MyThread

{

public int Count;

public Thread Thrd;

public MyThread(string name)

{

Count = 0;

Thrd = new Thread(this.Run);

Thrd.Name = name; //name of thread

Thrd.Start(); //begin thread

}

//point of start

void Run()

{

Console.WriteLine(Thrd.Name + " has begun.");

do

{

Thread.Sleep(500);

Console.WriteLine("In thread " + Thrd.Name + ", Count = " + Count);

Count++;

} while (Count < 10);

Console.WriteLine(Thrd.Name + " has ended.");

}

}

class MultiThreadImproved

{

static void Main()

{

Console.WriteLine("Main thread has begun.");

//first make object of MyThread

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

do

{

Console.Write(".");

Thread.Sleep(100);

} while (mt.Count != 10);

Console.WriteLine("Main thread has ended.");

}

}

Эта версия программы дает такой же результат, как и предыдущая. Обратите внимание

на то, что объект потока сохраняется в переменной Thrd из класса MyThread.

**Создание нескольких потоков**

В предыдущих примерах программ был создан лишь один порожденный поток.

Но в программе можно породить столько потоков, сколько потребуется. Например,

в следующей программе создаются три порожденных потока.

(***glava23\_3***)

using System.Threading;

class MyThread

{

public int Count;

public Thread Thrd;

public MyThread(string name)

{

Count = 0;

Thrd = new Thread(this.Run);

Thrd.Name = name;

Thrd.Start();

}

//poin of start

void Run()

{

Console.WriteLine(Thrd.Name + " has started.");

do

{

Thread.Sleep(500);

Console.WriteLine("In thread " + Thrd.Name + ", Count = " + Count);

Count++;

} while (Count < 10);

Console.WriteLine(Thrd.Name + " has ended.");

}

}

class MoreThreads

{

static void Main()

{

Console.WriteLine("Main thread started.");

//build 3 threads

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

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

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

do

{

Console.Write('.');

Thread.Sleep(100);

} while (mt1.Count < 10 ||

mt2.Count < 10 ||

mt3.Count < 10);

Console.WriteLine("Main thread ended.");

}

}

Как видите, после того как все три потока начнут выполняться, они будут совместно

использовать ЦП. Приведенный выше результат может отличаться в зависимости от

среды выполнения, операционной системы и других внешних факторов, влияющих на

выполнение программы.

**Определение момента окончания потока**

Нередко оказывается полезно знать, когда именно завершается поток. В предыдущих

примерах программ для этой цели отслеживалось значение переменной Count.

Но ведь это далеко не лучшее и не совсем пригодное для обобщения решение. Правда,

в классе Thread имеются два других средства для определения момента окончания

потока. С этой целью можно, прежде всего, опросить доступное только для чтения

свойство IsAlive, определяемое следующим образом.

public bool IsAlive { get; }

Свойство IsAlive возвращает логическое значение true, если поток, для которого

оно вызывается, по-прежнему выполняется. Для "опробования" свойства IsAlive

подставьте приведенный ниже фрагмент кода вместо кода в классе MoreThread из

предыдущей версии многопоточной программы, как показано ниже.

(***glava23\_4***)

// Использовать свойство IsAlive для отслеживания момента окончания потоков.

class MoreThreads

{

static void Main()

{

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

// Сконструировать три потока.

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

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

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

do

{

Console.Write(".");

Thread.Sleep(100);

} while (mt1.Thrd.IsAlive &&

mt2.Thrd.IsAlive &&

mt3.Thrd.IsAlive);

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

}

}

При выполнении этой версии программы результат получается таким же, как

и прежде. Единственное отличие заключается в том, что в ней используется свойство

IsAlive для отслеживания момента окончания порожденных потоков.

Еще один способ отслеживания момента окончания состоит в вызове метода

Join(). Ниже приведена его простейшая форма.

public void Join()

Метод Join() ожидает до тех пор, пока поток, для которого он был вызван, не

завершится. Его имя отражает принцип ожидания до тех пор, пока вызывающий поток

не *присоединится* к вызванному методу. Если же данный поток не был начат, то

генерируется исключение ThreadStateException. В других формах метода Join()

можно указать максимальный период времени, в течение которого следует ожидать

завершения указанного потока.

В приведенном ниже примере программы метод Join() используется для того,

чтобы основной поток завершился последним.

(***glava23\_5***)

using System.Threading;

class MyThread

{

public int Count;

public Thread Thrd;

public MyThread(string name)

{

Count = 0;

Thrd = new Thread(this.Run);

Thrd.Name = name;

Thrd.Start();

}

//point of start

void Run()

{

Console.WriteLine(Thrd.Name + " started.");

do

{

Thread.Sleep(500);

Console.WriteLine("In thread " + Thrd.Name + ", Count = " + Count);

Count++;

} while (Count < 10);

Console.WriteLine(Thrd.Name + " ended.");

}

}

//use Join() to wait until

//threads to be done

class JoinThreads

{

static void Main()

{

Console.WriteLine("Main thread started.");

//build 3 threads

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

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

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

mt1.Thrd.Join();

Console.WriteLine("Thread #1 joined.");

mt2.Thrd.Join();

Console.WriteLine("Thread #2 joined.");

mt3.Thrd.Join();

Console.WriteLine("Thread #3 joined.");

Console.WriteLine("Main thread ended.");

}

}

Как видите, выполнение потоков завершилось после возврата из последовательного

ряда вызовов метода Join().

**Передача аргумента потоку**

Первоначально в среде .NET Framework нельзя было передавать аргумент потоку,

когда он начинался, поскольку у метода, служившего в качестве точки входа в поток, не

могло быть параметров. Если же потоку требовалось передать какую-то информацию,

то к этой цели приходилось идти различными обходными путями, например использовать

общую переменную. Но этот недостаток был впоследствии устранен, и теперь

аргумент может быть передан потоку. Для этого придется воспользоваться другими

формами метода Start(), конструктора класса Thread, а также метода, служащего в

качестве точки входа в поток.

Аргумент передается потоку в следующей форме метода Start().

public void Start(object параметр)

Объект, указываемый в качестве аргумента *параметр,* автоматически передается

методу, выполняющему роль точки входа в поток. Следовательно, для того чтобы передать

аргумент потоку, достаточно передать его методу Start().

Для применения параметризированной формы метода Start() потребуется следующая

форма конструктора класса Thread:

public Thread(ParameterizedThreadStart запуск)

где *запуск* обозначает метод, вызываемый с целью начать выполнение потока.

Обратите внимание на то, что в этой форме конструктора запуск имеет тип

ParameterizedThreadStart, а не ThreadStart, как в форме, использовавшейся

в предыдущих примерах. В данном случае ParameterizedThreadStart является делегатом,

объявляемым следующим образом.

public delegate void ParameterizedThreadStart(object obj)

Как видите, этот делегат принимает аргумент типа object. Поэтому для правильного

применения данной формы конструктора класса Thread у метода, служащего

в качестве точки входа в поток, должен быть параметр типа object.

В приведенном ниже примере программы демонстрируется передача аргумента

потоку.

(***glava23\_6***)

using System.Threading;

class MyThread

{

public int Count;

public Thread Thrd;

//pay attention, constructor of MyThread,

//also gets value type int

public MyThread(string name, int num)

{

Count = 0;

//Call constructor type ParameterizedThreadStart

//explicit just to see how

Thrd = new Thread(this.Run);

Thrd.Name = name;

//here is variable num passed to method Start()

//as argument

Thrd.Start(num);

}

//pay attention, this form of method Run()

//takes parameter type object

void Run(object num)

{

Console.WriteLine(Thrd.Name + " started from " + num);

do

{

Thread.Sleep(500);

Console.WriteLine("In thread " + Thrd.Name + ", Count = " + Count);

Count++;

} while (Count < (int)num);

Console.WriteLine(Thrd.Name + " done.");

}

}

class PassArgDemo

{

static void Main()

{

//pay attention that numbers of repeat

//passed to this two objects type MyThread

MyThread mt = new MyThread("Thread #1", 5);

MyThread mt2 = new MyThread("Thread #2", 3);

do

{

Thread.Sleep(100);

} while (mt.Thrd.IsAlive | mt2.Thrd.IsAlive);

Console.WriteLine("Main thread done.");

}

}

Как следует из приведенного выше результата, первый поток повторяется пять раз,

а второй — три раза. Число повторений указывается в конструкторе класса MyThread

и затем передается методу Run(), служащему в качестве точки входа в поток, с помощью

параметризированной формы ParameterizedThreadStart метода Start().

**Свойство IsBackground**

Как упоминалось выше, в среде .NET Framework определены две разновидности

потоков: приоритетный и фоновый. Единственное отличие между ними заключается

в том, что процесс не завершится до тех пор, пока не окончится приоритетный поток,

тогда как фоновые потоки завершаются автоматически по окончании всех приоритетных

потоков. По умолчанию создаваемый поток становится приоритетным. Но его

можно сделать фоновым, используя свойство IsBackground, определенное в классе

Thread, следующим образом.

Для того чтобы сделать поток фоновым, достаточно присвоить логическое значение

true свойству IsBackground. А логическое значение false указывает на то, что поток

является приоритетным.

**Приоритеты потоков**

У каждого потока имеется свой приоритет, который отчасти определяет, насколько

часто поток получает доступ к ЦП. Вообще говоря, низкоприоритетные потоки получают

доступ к ЦП реже, чем высокоприоритетные. Таким образом, в течение заданного

промежутка времени низкоприоритетному потоку будет доступно меньше времени

ЦП, чем высокоприоритетному. Как и следовало ожидать, время ЦП, получаемое потоком,

оказывает определяющее влияние на характер его выполнения и взаимодействия

с другими потоками, исполняемыми в настоящий момент в системе.

Следует иметь в виду, что, помимо приоритета, на частоту доступа потока к ЦП

оказывают влияние и другие факторы. Так, если высокоприоритетный поток ожидает

доступа к некоторому ресурсу, например для ввода с клавиатуры, он блокируется,

а вместо него выполняется низкоприоритетный поток. В подобной ситуации низкоприоритетный

поток может получать доступ к ЦП чаще, чем высокоприоритетный

поток в течение определенного периода времени. И наконец, конкретное планирование

задач на уровне операционной системы также оказывает влияние на время ЦП,

выделяемое для потока.

Когда порожденный поток начинает выполняться, он получает приоритет, устанавливаемый

по умолчанию. Приоритет потока можно изменить с помощью свойства

Priority, являющегося членом класса Thread. Ниже приведена общая форма данного

свойства:

public ThreadPriority Priority { get; set; }

где ThreadPriority обозначает перечисление, в котором определяются приведенные

ниже значения приоритетов.

ThreadPriority.Highest

ThreadPriority.AboveNormal

ThreadPriority.Normal

ThreadPriority.BelowNormal

ThreadPriority.Lowest

По умолчанию для потока устанавливается значение приоритета ThreadPriority.

Normal.

Для того чтобы стало понятнее влияние приоритетов на исполнение потоков, обратимся

к примеру, в котором выполняются два потока: один с более высоким приоритетом.

Оба потока создаются в качестве экземпляров объектов класса MyThread.

В методе Run() организуется цикл, в котором подсчитывается определенное число

повторений. Цикл завершается, когда подсчет достигает величины 1000000000 или когда

статическая переменная stop получает логическое значение true. Первоначально

переменная stop получает логическое значение false. В первом потоке, где производится

подсчет до 1000000000, устанавливается логическое значение true переменной

stop. В силу этого второй поток оканчивается на следующем своем интервале

времени. На каждом шаге цикла строка в переменной currentName проверяется на

наличие имени исполняемого потока. Если имена потоков не совпадают, это означает,

что произошло переключение исполняемых задач. Всякий раз, когда происходит

переключение задач, имя нового потока отображается и присваивается переменной

currentName. Это дает возможность отследить частоту доступа потока к ЦП. По окончании

обоих потоков отображается число повторений цикла в каждом из них.

(***glava23\_6***)

using System.Threading;

class MyThread

{

public int Count;

public Thread Thrd;

static bool stop = false;

static string currentName;

//build new thread. Pay attention that,

//this constructor is not starting thread yet

public MyThread(string name)

{

Count = 0;

Thrd = new Thread(this.Run);

Thrd.Name = name;

currentName = name;

}

//start new thread

void Run()

{

Console.WriteLine("Thread " + Thrd.Name + " started.");

do

{

Count++;

if (currentName != Thrd.Name)

{

currentName = Thrd.Name;

Console.WriteLine("In thread " + currentName);

}

} while (stop == false && Count < 1000000000);

stop = true;

Console.WriteLine("Thread " + Thrd.Name + " done.");

}

}

class PriorityDemo

{

static void Main()

{

MyThread mt1 = new MyThread("High priority thread");

MyThread mt2 = new MyThread("Low priority thread");

//set priorities for threads

mt1.Thrd.Priority = ThreadPriority.AboveNormal;

mt2.Thrd.Priority = ThreadPriority.BelowNormal;

//run threads

mt1.Thrd.Start();

mt2.Thrd.Start();

mt1.Thrd.Join();

mt2.Thrd.Join();

Console.WriteLine();

Console.WriteLine("Thread " + mt1.Thrd.Name + " count until " + mt1.Count);

Console.WriteLine("Thread " + mt2.Thrd.Name + " count until " + mt2.Count);

}

}

Судя по результату, высокоприоритетный поток получил около больший % всего времени,

которое было выделено для выполнения этой программы. Разумеется, конкретный результат

может отличаться в зависимости от быстродействия ЦП и числа других задач,

решаемых в системе, а также от используемой версии Windows.

Многопоточный код может вести себя по-разному в различных средах, поэтому

никогда не следует полагаться на результаты его выполнения только в одной среде.

Так, было бы ошибкой полагать, что низкоприоритетный поток из приведенного выше

примера будет всегда выполняться лишь в течение небольшого периода времени до

тех пор, пока не завершится высокоприоритетный поток. В другой среде высокоприоритетный

поток может, например, завершиться еще до того, как низкоприоритетный

поток выполнится хотя бы один раз.

**Синхронизация**

Когда используется несколько потоков, то иногда приходится координировать действия

двух или более потоков. Процесс достижения такой координации называется

*синхронизацией.* Самой распространенной причиной применения синхронизации служит

необходимость разделять среди двух или более потоков общий ресурс, который

может быть одновременно доступен только одному потоку. Например, когда в одном

потоке выполняется запись информации в файл, второму потоку должно быть запрещено

делать это в тот же самый момент времени. Синхронизация требуется и в том

случае, если один поток ожидает событие, вызываемое другим потоком. В подобной

ситуации требуются какие-то средства, позволяющие приостановить один из потоков

до тех пор, пока не произойдет событие в другом потоке. После этого ожидающий поток

может возобновить свое выполнение.

В основу синхронизации положено понятие *блокировки,* посредством которой организуется

управление доступом к кодовому блоку в объекте. Когда объект заблокирован

одним потоком, остальные потоки не могут получить доступ к заблокированному

кодовому блоку. Когда же блокировка снимается одним потоком, объект становится

доступным для использования в другом потоке.

Средство блокировки встроено в язык С#. Благодаря этому все объекты могут быть

синхронизированы. Синхронизация организуется с помощью ключевого слова lock.

Она была предусмотрена в C# с самого начала, и поэтому пользоваться ею намного

проще, чем кажется на первый взгляд. В действительности синхронизация объектов во

многих программах на С# происходит практически незаметно.

Ниже приведена общая форма блокировки:

lock(lockObj) {

// синхронизируемые операторы

}

где *lockObj* обозначает ссылку на синхронизируемый объект. Если же требуется синхронизировать

только один оператор, то фигурные скобки не нужны. Оператор lock

гарантирует, что фрагмент кода, защищенный блокировкой для данного объекта, будет

использоваться только в потоке, получающем эту блокировку. А все остальные потоки

блокируются до тех пор, пока блокировка не будет снята. Блокировка снимается

по завершении защищаемого ею фрагмента кода.

Блокируемым считается такой объект, который представляет синхронизируемый

ресурс. В некоторых случаях им оказывается экземпляр самого ресурса или же произвольный

экземпляр объекта, используемого для синхронизации. Следует, однако,

иметь в виду, что блокируемый объект не должен быть общедоступным, так как в противном

случае он может быть заблокирован из другого, неконтролируемого в программе

фрагмента кода и в дальнейшем вообще не разблокируется. В прошлом для

блокировки объектов очень часто применялась конструкция lock(this). Но она пригодна

только в том случае, если this является ссылкой на закрытый объект. В связи с

возможными программными и концептуальными ошибками, к которым может привести

конструкция lock(this), применять ее больше не рекомендуется. Вместо нее

лучше создать закрытый объект, чтобы затем заблокировать его. Именно такой подход

принят в примерах программ, приведенных далее в этой главе. Но в унаследованном

коде C# могут быть обнаружены примеры применения конструкции lock(this).

В одних случаях такой код оказывается безопасным, а в других — требует изменений

во избежание серьезных осложнений при его выполнении.

В приведенной ниже программе синхронизация демонстрируется на примере

управления доступом к методу SumIt(), суммирующему элементы целочисленного

массива.

(***glava23\_7***)

using System.Threading;

class SumArray

{

int sum;

object lockOn = new object();//closed object, accessable for lock

public int SumIt(int[] nums)

{

lock (lockOn)//lock all methods

{

sum = 0;//set default value of sum

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

{

sum += nums[i];

Console.WriteLine("Current summ for thread" +

Thread.CurrentThread.Name + " = " + sum);

Thread.Sleep(10);//access change task

}

return sum;

}

}

}

class MyThread

{

public Thread Thrd;

int[] a;

int answer;

//create one object of SumArray for all

//MyThread objects

static SumArray sa = new SumArray();

//build new thread

public MyThread(string name, int[] nums)

{

a = nums;

Thrd = new Thread(this.Run);

Thrd.Name = name;

Thrd.Start(); //start thread

}

//start new thread

void Run()

{

Console.WriteLine(Thrd.Name + " started.");

answer = sa.SumIt(a);

Console.WriteLine("Summ for thread " + Thrd.Name + " = " + answer);

Console.WriteLine(Thrd.Name + " done.");

}

}

class Sync

{

static void Main()

{

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

MyThread mt1 = new MyThread("Thread #1", a);

MyThread mt2 = new MyThread("Thread #2", a);

mt1.Thrd.Join();

mt2.Thrd.Join();

}

}

Как следует из приведенного выше результата, в обоих потоках правильно подсчитывается

сумма, равная 15.

Рассмотрим эту программу более подробно. Сначала в ней создаются три класса.

Первым из них оказывается класс SumArray, в котором определяется метод SumIt(),

суммирующий элементы целочисленного массива. Вторым создается класс MyThread,

в котором используется статический объект sa типа SumArray. Следовательно, единственный

объект типа SumArray используется всеми объектами типа MyThread. С помощью

этого объекта получается сумма элементов целочисленного массива. Обратите

внимание на то, что текущая сумма запоминается в поле sum объекта типа SumArray.

Поэтому если метод SumIt() используется параллельно в двух потоках, то оба потока

попытаются обратиться к полю sum, чтобы сохранить в нем текущую сумму. А поскольку

это может привести к ошибкам, то доступ к методу SumIt() должен быть

синхронизирован. И наконец, в третьем классе, Sync, создаются два потока, в которых

подсчитывается сумма элементов целочисленного массива.

Оператор lock в методе SumIt() препятствует одновременному использованию

данного метода в разных потоках. Обратите внимание на то, что в операторе lock объект

lockOn используется в качестве синхронизируемого. Это закрытый объект, предназначенный

исключительно для синхронизации. Метод Sleep() намеренно вызывается

для того, чтобы произошло переключение задач, хотя в данном случае это невозможно.

Код в методе SumIt() заблокирован, и поэтому он может быть одновременно использован

только в одном потоке. Таким образом, когда начинает выполняться второй

порожденный поток, он не сможет войти в метод SumIt() до тех пор, пока из него не

выйдет первый порожденный поток. Благодаря этому гарантируется получение правильного

результата.

Для того чтобы полностью уяснить принцип действия блокировки, попробуйте

удалить из рассматриваемой здесь программы тело метода lock().

В итоге метод SumIt() перестанет быть синхронизированным, а следовательно, он может параллельно использоваться в любом числе потоков для одного и того же объекта. Поскольку

текущая сумма сохраняется в поле sum, она может быть изменена в каждом потоке,

вызывающем метод SumIt(). Это означает, что если два потока одновременно вызывают

метод SumIt() для одного и того же объекта, то конечный результат получается неверным, поскольку содержимое поля sum отражает смешанный результат суммирования

в обоих потоках. В качестве примера ниже приведен результат выполнения

рассматриваемой здесь программы после снятия блокировки с метода SumIt().

Ниже подведены краткие итоги использования блокировки.

**•** Если блокировка любого заданного объекта получена в одном потоке, то после

блокировки объекта она не может быть получена в другом потоке.

• Остальным потокам, пытающимся получить блокировку того же самого объекта,

придется ждать до тех пор, пока объект не окажется в разблокированном

состоянии.

• Когда поток выходит из заблокированного фрагмента кода, соответствующий

объект разблокируется.

**Другой подход к синхронизации потоков**

Несмотря на всю простоту и эффективность блокировки кода метода, как показано

в приведенном выше примере, такое средство синхронизации оказывается пригодным

далеко не всегда. Допустим, что требуется синхронизировать доступ к методу класса,

который был создан кем-то другим и сам не синхронизирован. Подобная ситуация

вполне возможна при использовании чужого класса, исходный код которого недоступен.

В этом случае оператор lock нельзя ввести в соответствующий метод чужого

класса. Как же тогда синхронизировать объект такого класса? К счастью, этот вопрос

разрешается довольно просто: доступ к объекту может быть заблокирован из внешнего

кода по отношению к данному объекту, для чего достаточно указать этот объект

в операторе lock. В качестве примера ниже приведен другой вариант реализации

предыдущей программы. Обратите внимание на то, что код в методе SumIt() уже не

является заблокированным, а объект lockOn больше не объявляется. Вместо этого вызовы

метода SumIt() блокируются в классе MyThread.

(***glava23\_8***)

using System.Threading;

class SumArray

{

int sum;

public int SumIt(int[] nums)

{

sum = 0;//set default value of sum

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

{

sum += nums[i];

Console.WriteLine("Current summ for thread" +

Thread.CurrentThread.Name + " = " + sum);

Thread.Sleep(10);//access change task

}

return sum;

}

}

class MyThread

{

public Thread Thrd;

int[] a;

int answer;

//create one object of SumArray for all

//MyThread objects

static SumArray sa = new SumArray();

//build new thread

public MyThread(string name, int[] nums)

{

a = nums;

Thrd = new Thread(this.Run);

Thrd.Name = name;

Thrd.Start(); //start thread

}

//start new thread

void Run()

{

Console.WriteLine(Thrd.Name + " started.");

//lock method SumIt()

lock (sa) answer = sa.SumIt(a);

Console.WriteLine("Summ for thread " + Thrd.Name + " = " + answer);

Console.WriteLine(Thrd.Name + " done.");

}

}

class Sync

{

static void Main()

{

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

MyThread mt1 = new MyThread("Thread #1", a);

MyThread mt2 = new MyThread("Thread #2", a);

mt1.Thrd.Join();

mt2.Thrd.Join();

}

}

В данной программе блокируется вызов метода sa.SumIt(), а не сам метод

SumIt(). Ниже приведена соответствующая строка кода, в которой осуществляется

подобная блокировка.

//lock method SumIt()

lock (sa) answer = sa.SumIt(a);

Объект sa является закрытым, и поэтому он может быть благополучно заблокирован.

При таком подходе к синхронизации потоков данная программа дает такой же

правильный результат, как и при первоначальном подходе.

**Класс Monitor и блокировка**

Ключевое слово lock на самом деде служит в C# быстрым способом доступа к

средствам синхронизации, определенным в классе Monitor, который находится в пространстве

имен System.Threading. В этом классе определен, в частности, ряд методов

для управления синхронизацией. Например, для получения блокировки объекта вызывается

метод Enter(), а для снятия блокировки — метод Exit(). Ниже приведены

общие формы этих методов:

public static void Enter(object obj)

public static void Exit(object obj)

где *obj* обозначает синхронизируемый объект. Если же объект недоступен, то после

вызова метода Enter() вызывающий поток ожидает до тех пор, пока объект не станет

доступным. Тем не менее методы Enter() и Exit() применяются редко, поскольку

оператор lock автоматически предоставляет эквивалентные средства синхронизации

потоков. Именно поэтому оператор lock оказывается "более предпочтительным" для

получения блокировки объекта при программировании на С#.

Впрочем, один метод из класса Monitor может все же оказаться полезным. Это

метод TryEnter(), одна из общих форм которого приведена ниже.

public static bool TryEnter(object obj)

Этот метод возвращает логическое значение true, если вызывающий поток получает

блокировку для объекта *obj,* а иначе он возвращает логическое значение false.

Но в любом случае вызывающему потоку придется ждать своей очереди. С помощью

метода TryEnter() можно реализовать альтернативный вариант синхронизации потоков,

если требуемый объект временно недоступен.

Кроме того, в классе Monitor определены методы Wait(), Pulse() и PulseAll(),

которые рассматриваются в следующем разделе.

**Сообщение между потоками с помощью методов**

**Wait(), Pulse() и PulseAll()**

Рассмотрим следующую ситуацию. Поток *Т* выполняется в кодовом блоке lock,

и ему требуется доступ к ресурсу *R,* который временно недоступен. Что же тогда делать

потоку *Т*? Если поток *Т* войдет в организованный в той или иной форме цикл опроса,

ожидая освобождения ресурса *R,* то тем самым он свяжет соответствующий объект,

блокируя доступ к нему других потоков. Это далеко не самое оптимальное решение,

поскольку оно лишает отчасти преимуществ программирования для многопоточной

среды. Более совершенное решение заключается в том, чтобы временно освободить

объект и тем самым дать возможность выполняться другим потокам. Такой подход

основывается на некоторой форме сообщения между потоками, благодаря которому

один поток может уведомлять другой о том, что он заблокирован и что другой поток

может возобновить свое выполнение. Сообщение между потоками организуется в C# с

помощью методов Wait(), Pulse() и PulseAll().

Методы Wait(), Pulse() и PulseAll() определены в классе Monitor и могут

вызываться только из заблокированного фрагмента блока. Они применяются следующим

образом. Когда выполнение потока временно заблокировано, он вызывает метод

Wait(). В итоге поток переходит в состояние ожидания, а блокировка с соответствующего

объекта снимается, что дает возможность использовать этот объект в другом потоке.

В дальнейшем ожидающий поток активизируется, когда другой поток войдет в

аналогичное состояние блокировки, и вызывает метод Pulse() или PulseAll(). При

вызове метода Pulse() возобновляется выполнение первого потока, ожидающего своей

очереди на получение блокировки. А вызов метода PulseAll() сигнализирует о

снятии блокировки всем ожидающим потокам.

Ниже приведены две наиболее часто используемые формы метода Wait().

public static bool Wait(object obj)

public static bool Wait(object obj, int миллисекунд\_простоя)

В первой форме ожидание длится вплоть до уведомления об освобождении объекта,

а во второй форме — как до уведомления об освобождении объекта, так и до истечения

периода времени, на который указывает количество *миллисекунд\_простоя.*

В обеих формах *obj* обозначает объект, освобождение которого ожидается.

Ниже приведены общие формы методов Pulse() и PulseAll():

public static void Pulse(object obj)

public static void PulseAll(object obj)

где *obj* обозначает освобождаемый объект.

Если методы Wait(), Pulse() и PulseAll() вызываются из кода, находящегося

за пределами синхронизированного кода, например из блока lock, то генерируется

исключение SynchronizationLockException.

**Пример использования методов Wait() и Pulse()**

Для того чтобы стало понятнее назначение методов Wait() и Pulse(), рассмотрим

пример программы, имитирующей тиканье часов и отображающей этот процесс на

экране словами "тик" и "так". Для этой цели в программе создается класс TickTock,

содержащий два следующих метода: Tick() и Тоск(). Метод Tick() выводит на

экран слово "тик", а метод Тоск() — слово "так". Для запуска часов далее в программе

создаются два потока: один из них вызывает метод Tick(), а другой — метод Тоск().

Преследуемая в данном случае цель состоит в том, чтобы оба потока выполнялись, поочередно

выводя на экран слова "тик" и "так", из которых образуется повторяющийся

ряд "тик-так", имитирующий ход часов.

(***glava23\_9***)

using System.Threading;

class TickTock

{

object lockOn = new object();

public void Tick(bool running)

{

lock(lockOn)

{

if(!running)//stop clocks

{

Monitor.Pulse(lockOn);//warn any waiting threads

return;

}

Console.Write(" Tick ");

Monitor.Pulse(lockOn); //permit to do Tock()

Monitor.Wait(lockOn); //waiting for Tock() to be done

}

}

public void Tock(bool running)

{

lock(lockOn)

{

if(!running)//stop clocks

{

Monitor.Pulse(lockOn);//warn any wating threads

return;

}

Console.Write(" Tock ");

Monitor.Pulse(lockOn); //permit to do Tick()

Monitor.Wait(lockOn); //waiting for Tick() to be done

}

}

}

class MyThread

{

public Thread Thrd;

TickTock ttOb;

//build new thread

public MyThread(string name, TickTock tt)

{

Thrd = new Thread(this.Run);

ttOb = tt;

Thrd.Name = name;

Thrd.Start();

}

//start new thread

void Run()

{

if(Thrd.Name == "Tick")

{

for (int i = 0; i < 5; i++) ttOb.Tick(true);

ttOb.Tick(false);

}

else

{

for (int i = 0; i < 5; i++) ttOb.Tock(true);

ttOb.Tock(false);

}

}

}

class TickingClock

{

static void Main()

{

TickTock tt = new TickTock();

MyThread mt1 = new MyThread("Tick", tt);

MyThread mt2 = new MyThread("Tock", tt);

mt1.Thrd.Join();

mt2.Thrd.Join();

Console.WriteLine("Clock has been stoped.");

}

}

Рассмотрим эту программу более подробно. В методе Main() создается объект tt

типа TickTock, который используется для запуска двух потоков на выполнение. Если

в методе Run() из класса MyThread обнаруживается имя потока Tick, соответствующее

ходу часов "тик", то вызывается метод Tick(). А если это имя потока Тоск, соответствующее

ходу часов "так", то вызывается метод Тоск(). Каждый из этих методов

вызывается пять раз подряд с передачей логического значения true в качестве apiy-

мента. Часы идут до тех пор, пока этим методам передается логическое значение true,

и останавливаются, как только передается логическое значение false.

Самая важная часть рассматриваемой здесь программы находится в методах

Tick() и Тоск(). Начнем с метода Tick(), код которого для удобства приводится

ниже.

public void Tick(bool running)

{

lock(lockOn)

{

if(!running)//stop clocks

{

Monitor.Pulse(lockOn);//warn any waiting threads

return;

}

Console.Write(" Tick ");

Monitor.Pulse(lockOn); //permit to do Tock()

Monitor.Wait(lockOn); //waiting for Tock() to be done

}

}

Прежде всего обратите внимание на код метода Tick() в блоке lock. Напомним,

что методы Wait() и Pulse() могут использоваться только в синхронизированных

блоках кода. В начале метода Tick() проверяется значение текущего параметра, которое

служит явным признаком остановки часов. Если это логическое значение false,

то часы остановлены. В этом случае вызывается метод Pulse(), разрешающий выполнение

любого потока, ожидающего своей очереди. Мы еще вернемся к этому моменту

в дальнейшем. Если же часы идут при выполнении метода Tick(), то на экран выводится

слово "тик" с пробелом, затем вызывается метод Pulse(), а после него — метод

Wait(). При вызове метода Pulse() разрешается выполнение потока для того же

самого объекта, а при вызове метода Wait() выполнение метода Tick() приостанавливается

до тех пор, пока метод Pulse() не будет вызван из другого потока. Таким

образом, когда вызывается метод Tick(), отображается одно слово "тик" с пробелом,

разрешается выполнение другого потока, а затем выполнение данного метода приостанавливается.

Метод Tock() является точной копией метода Tick(), за исключением того, что

он выводит на экран слово "так". Таким образом, при входе в метод Tock() на экран

выводится слово "так", вызывается метод Pulse(), а затем выполнение метода Tock()

приостанавливается. Методы Tick() и Tock() можно рассматривать как поочередно

сменяющие друг друга, т.е. они взаимно синхронизированы.

Когда часы остановлены, метод Pulse() вызывается для того, чтобы обеспечить

успешный вызов метода Wait(). Напомним, что метод Wait() вызывается в обоих

методах, Tick() и Tock(), после вывода соответствующего слова на экран. Но дело в

том, что когда часы остановлены, один из этих методов все еще находится в состоянии

ожидания. Поэтому завершающий вызов метода Pulse() требуется, чтобы выполнить

ожидающий метод до конца. В качестве эксперимента попробуйте удалить этот

вызов метода Pulse() и понаблюдайте за тем, что при этом произойдет. Вы сразу же

обнаружите, что программа "зависает", и для выхода из нее придется нажать комбинацию

клавиш <Ctrl+C>. Дело в том, что когда метод Wait() вызывается в последнем

вызове метода Tock(), соответствующий ему метод Pulse() не вызывается, а значит,

выполнение метода Tock() оказывается незавершенным, и он ожидает своей очереди

до бесконечности.

**Взаимоблокировка и состояние гонки**

При разработке многопоточных программ следует быть особенно внимательным,

чтобы избежать взаимоблокировки и состояний гонок. *Взаимоблокировка,* как подразумевает

само название, — это ситуация, в которой один поток ожидает определенных

действий от другого потока, а другой поток, в свою очередь, ожидает чего-то от первого

потока. В итоге оба потока приостанавливаются, ожидая друг друга, и ни один из

них не выполняется. Эта ситуация напоминает двух слишком вежливых людей, каждый

из которых настаивает на том, чтобы другой прошел в дверь первым!

На первый взгляд избежать взаимоблокировки нетрудно, но на самом деле не все так

просто, ведь взаимоблокировка может возникать окольными путями. В качестве примера

рассмотрим класс TickTock из предыдущей программы. Как пояснялось выше,

в отсутствие завершающего вызова метода Pulse() из метода Tick() или Тосk() тот

или другой будет ожидать до бесконечности, что приведет к "зависанию" программы

вследствие взаимоблокировки. Зачастую причину взаимоблокировки не так-то просто

выяснить, анализируя исходный код программы, поскольку параллельно действующие

процессы могут взаимодействовать довольно сложным образом во время выполнения.

Для исключения взаимоблокировки требуется внимательное программирование и

тщательное тестирование. В целом, если многопоточная программа периодически "зависает",

то наиболее вероятной причиной этого является взаимоблокировка.

*Состояние гонки* возникает в том случае, когда два потока или больше пытаются

одновременно получить доступ к общему ресурсу без должной синхронизации. Так,

в одном потоке может сохраняться значение в переменной, а в другом — инкрементироваться

текущее значение этой же переменной. В отсутствие синхронизации конечный

результат будет зависеть от того, в каком именно порядке выполняются потоки:

инкрементируется ли значение переменной во втором потоке или же оно сохраняется

в первом. О подобной ситуации говорят, что потоки "гоняются друг за другом", причем

конечный результат зависит от того, какой из потоков завершится первым. Возникающее

состояние гонок, как и взаимоблокировку, непросто обнаружить. Поэтому его

лучше предотвратить, синхронизируя должным образом доступ к общим ресурсам

при программировании.

**Применение атрибута MethodImplAttribute**

Метод может быть полностью синхронизирован с помощью атрибута

MethodImplAttribute. Такой подход может стать альтернативой оператору

lock в тех случаях, когда метод требуется заблокировать полностью. Атрибут

MethodImplAttribute определен в пространстве имен Sуstem.Runtime.

CompilerServices. Ниже приведен конструктор, применяемый для подобной

синхронизации:

public MethodImplAttribute(MethodImplOptions methodImplOptions)

где *methodImplOptions* обозначает атрибут реализации. Для синхронизации метода

достаточно указать атрибут MethodImplOptions.Synchronized. Этот атрибут вызывает

блокировку всего метода для текущего экземпляра объекта, доступного по ссылке

this. Если же метод относится к типу static, то блокируется его тип. Поэтому данный

атрибут непригоден для применения в открытых объектах иди классах.

Ниже приведена еще одна версия программы, имитирующей тиканье часов, с переделанным

вариантом класса TickTock, в котором атрибут MethodImplOptions обеспечивает

должную синхронизацию.

(***glava23\_10***)

using System.Threading;

using System.Runtime.CompilerServices;

//variant of class TickTock changed to use attribute

// MethodImplOptions.Synchronized

class TickTock

{

//next attribute fully synchronize method Tick()

[MethodImplAttribute(MethodImplOptions.Synchronized)]

public void Tick(bool running)

{

if (!running)//stop clocks

{

Monitor.Pulse(this);//warn any waiting threads

return;

}

Console.Write("Tick ");

Monitor.Pulse(this); //permit to do Tock()

Monitor.Wait(this); //waiting for Tock() to be done

}

//next attribute fully synchronize method Tock()

[MethodImplAttribute(MethodImplOptions.Synchronized)]

public void Tock(bool running)

{

if (!running)//stop clocks

{

Monitor.Pulse(this);//warn any wating threads

return;

}

Console.Write("Tock ");

Monitor.Pulse(this); //permit to do Tick()

Monitor.Wait(this); //waiting for Tick() to be done

}

}

class MyThread

{

public Thread Thrd;

TickTock ttOb;

//build new thread

public MyThread(string name, TickTock tt)

{

Thrd = new Thread(this.Run);

ttOb = tt;

Thrd.Name = name;

Thrd.Start();

}

//start new thread

void Run()

{

if (Thrd.Name == "Tick")

{

for (int i = 0; i < 5; i++) ttOb.Tick(true);

ttOb.Tick(false);

}

else

{

for (int i = 0; i < 5; i++) ttOb.Tock(true);

ttOb.Tock(false);

}

}

}

class TickingClock

{

static void Main()

{

TickTock tt = new TickTock();

MyThread mt1 = new MyThread("Tick", tt);

MyThread mt2 = new MyThread("Tock", tt);

mt1.Thrd.Join();

mt2.Thrd.Join();

Console.WriteLine("\nClock has been stoped.");

}

}

Эта версия программы дает такой же результат, как и предыдущая.

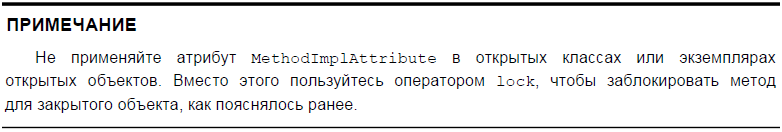
Синхронизируемый метод не определен в открытом классе и не вызывается

для открытого объекта, поэтому применение оператора lock или атрибута

MethodImplAttribute зависит от личных предпочтений. Ведь и тот и другой дает один

и тот же результат. Но поскольку ключевое слово lock относится непосредственно к

языку С#, то в примерах, приведенных в этой книге, предпочтение отдано именно ему.



**Применение мьютекса и семафора**

В большинстве случаев, когда требуется синхронизация, оказывается достаточно

и оператора lock. Тем не менее в некоторых случаях, как, например, при ограничении

доступа к общим ресурсам, более удобными оказываются механизмы синхронизации,

встроенные в среду .NET Framework. Ниже рассматриваются по порядку два таких механизма:

мьютекс и семафор.

**Мьютекс**

*Мьютекс* представляет собой взаимно исключающий синхронизирующий объект.

Это означает, что он может быть получен потоком только по очереди. Мьютекс

предназначен для тех ситуаций, в которых общий ресурс может быть одновременно

использован только в одном потоке. Допустим, что системный журнал совместно используется

в нескольких процессах, но только в одном из них данные могут записываться

в файл этого журнала в любой момент времени. Для синхронизации процессов

в данной ситуации идеально подходит мьютекс.

Мьютекс поддерживается в классе System.Threading.Mutex. У него имеется несколько

конструкторов. Ниже приведены два наиболее употребительных конструктора.

public Mutex()

public Mutex(bool initiallyOwned)

В первой форме конструктора создается мьютекс, которым первоначально никто не

владеет. А во второй форме исходным состоянием мьютекса завладевает вызывающий

поток, если параметр *initiallyOwned* имеет логическое значение true. В противном

случае мьютексом никто не владеет.

Для того чтобы получить мьютекс, в коде программы следует вызвать метод

WaitOne() для этого мьютекса. Метод WaitOne() наследуется классом Mutex от класса

Thread.WaitHandle. Ниже приведена его простейшая форма.

public bool WaitOne();

Метод WaitOne() ожидает до тех пор, пока не будет получен мьютекс, для которого

он был вызван. Следовательно, этот метод блокирует выполнение вызывающего потока

до тех пор, пока не станет доступным указанный мьютекс. Он всегда возвращает

логическое значение true.

Когда же в коде больше не требуется владеть мьютексом, он освобождается посредством

вызова метода ReleaseMutex(), форма которого приведена ниже.

public void ReleaseMutex()

В этой форме метод ReleaseMutex() освобождает мьютекс, для которого он был

вызван, что дает возможность другому потоку получить данный мьютекс.

Для применения мьютекса с целью синхронизировать доступ к общему ресурсу

упомянутые выше методы WaitOne() и ReleaseMutex() используются так, как показано

в приведенном ниже фрагменте кода.

Mutex myMtx = new Mutex();

// ...

myMtx.WaitOne(); // ожидать получения мьютекса

// Получить доступ к общему ресурсу.

myMtx.ReleaseMutex(); // освободить мьютекс

При вызове метода WaitOne() выполнение соответствующего потока приостанавливается

до тех пор, пока не будет получен мьютекс. А при вызове метода

ReleaseMutex() мьютекс освобождается и затем может быть получен другим потоком.

Благодаря такому подходу к синхронизации одновременный доступ к общему

ресурсу ограничивается только одним потоком.

В приведенном ниже примере программы описанный выше механизм синхронизации

демонстрируется на практике. В этой программе создаются два потока в виде

классов IncThread и DecThread, которым требуется доступ к общему ресурсу: переменной

SharedRes.Count. В потоке IncThread переменная SharedRes.Count

инкрементируется, а в потоке DecThread — декрементируется. Во избежание одновременного

доступа обоих потоков к общему ресурсу SharedRes.Count этот доступ

синхронизируется мьютексом Mtx, также являющимся членом класса SharedRes.

(***glava23\_11***)

//this class has common resourse(Count)

//and mutex (Mtx), controlint access to it

class SharedRes

{

public static int Count = 0;

public static Mutex Mtx = new Mutex();

}

//this thread variable SharedRes.Count increments

class IncThread

{

int num;

public Thread Thrd;

public IncThread(string name, int n)

{

Thrd = new Thread(this.Run);

num = n;

Thrd.Name = name;

Thrd.Start();

}

//point of start

void Run()

{

Console.WriteLine(Thrd.Name + " waiting for mutex.");

//Get mutex

SharedRes.Mtx.WaitOne();

Console.WriteLine(Thrd.Name + " gets mutex.");

do

{

Thread.Sleep(500);

SharedRes.Count++;

Console.WriteLine("In thread " + Thrd.Name

+ ", SharedRes.Count = " + SharedRes.Count);

num--;

} while (num > 0);

Console.WriteLine(Thrd.Name + " realeases mutex.");

//release mutex

SharedRes.Mtx.ReleaseMutex();

}

}

class DecThread

{

int num;

public Thread Thrd;

public DecThread(string name, int n)

{

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

num = n;

Thrd.Name = name;

Thrd.Start();

}

//point of start

void Run()

{

Console.WriteLine(Thrd.Name + " waiting for mutex.");

//get mutex

SharedRes.Mtx.WaitOne();

Console.WriteLine(Thrd.Name + " gets mutex.");

do

{

Thread.Sleep(500);

SharedRes.Count--;

Console.WriteLine("In thread " + Thrd.Name +

", SharedRes.Count = " + SharedRes.Count);

num--;

} while (num > 0);

Console.WriteLine(Thrd.Name + " realeases mutex.");

//release mutex

SharedRes.Mtx.ReleaseMutex();

}

}

class MutexDemo

{

static void Main()

{

//build two threads

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

Thread.Sleep(1);//allow increment thread to start

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

mt1.Thrd.Join();

mt2.Thrd.Join();

}

}

Как следует из приведенного выше результата, доступ к общему ресурсу (переменной

SharedRes.Count) синхронизирован, и поэтому значение данной переменной

может быть одновременно изменено только в одном потоке.

Для того чтобы убедиться в том, что мьютекс необходим для получения приведенного

выше результата, попробуйте закомментировать вызовы методов WaitOne()

и ReleaseMutex() в исходном коде рассматриваемой здесь программы.

Мьютекс, созданный в предыдущем примере, известен только тому процессу, который

его породил. Но мьютекс можно создать и таким образом, чтобы он был известен

где-нибудь еще. Для этого он должен быть именованным. Ниже приведены формы

конструктора, предназначенные для создания такого мьютекса.

public Mutex(bool initiallyOwned, string имя)

public Mutex(bool initiallyOwned, string имя, out bool createdNew)

В обеих формах конструктора *имя* обозначает конкретное имя мьютекса. Если в

первой форме конструктора параметр *initiallyOwned* имеет логическое значение

true, то владение мьютексом запрашивается. Но поскольку мьютекс может принадлежать

другому процессу на системном уровне, то для этого параметра лучше указать

логическое значение false. А после возврата из второй формы конструктора параметр

*createdNew* будет иметь логическое значение true, если владение мьютексом

было запрошено и получено, и логическое значение false, если запрос на владение

был отклонен. Существует и третья форма конструктора типа Mutex, в которой допускается

указывать управляющий доступом объект типа MutexSecurity. С помощью

именованных мьютексов можно синхронизировать взаимодействие процессов.

И последнее замечание: в потоке, получившем мьютекс, допускается делать один

иди несколько дополнительных вызовов метода WaitOne() перед вызовом метода

ReleaseMutex(), причем все эти дополнительные вызовы будут произведены успешно.

Это означает, что дополнительные вызовы метода WaitOne() не будут блокировать

поток, который уже владеет мьютексом. Но количество вызовов метода WaitOne()

должно быть равно количеству вызовов метода ReleaseMutex() перед освобождением

мьютекса.

**Семафор**

*Семафор* подобен мьютексу, за исключением того, что он предоставляет одновременный

доступ к общему ресурсу не одному, а нескольким потокам. Поэтому семафор

пригоден для синхронизации целого ряда ресурсов. Семафор управляет доступом к

общему ресурсу, используя для этой цели счетчик. Если значение счетчика больше

нуля, то доступ к ресурсу разрешен. А если это значение равно нулю, то доступ к ресурсу

запрещен. С помощью счетчика ведется подсчет количества *разрешений.* Следовательно,

для доступа к ресурсу поток должен получить разрешение от семафора.

Количество одновременно разрешаемых доступов указывается при создании

семафора. Так, если создать семафор, одновременно разрешающий только один доступ,

то такой семафор будет действовать как мьютекс.

Семафоры особенно полезны в тех случаях, когда общий ресурс состоит из группы

иди пуда ресурсов. Например, пул ресурсов может состоять из целого ряда сетевых

соединений, каждое из которых служит для передачи данных. Поэтому потоку, которому

требуется сетевое соединение, все равно, какое именно соединение он получит.

В данном случае семафор обеспечивает удобный механизм управления доступом к сетевым

соединениям.

Семафор реализуется в классе System.Threading.Semaphore, у которого имеется

несколько конструкторов. Ниже приведена простейшая форма конструктора данного

класса:

public Semaphore(int initialCount, int maximumCount)

где *initialCount* — это первоначальное значение для счетчика разрешений семафора,

т.е. количество первоначально доступных разрешений; *maximumCount —* максимальное

значение данного счетчика, т.е. максимальное количество разрешений, которые

может дать семафор.

Семафор применяется таким же образом, как и описанный ранее мьютекс. В целях

получения доступа к ресурсу в коде программы вызывается метод WaitOne() для

семафора. Этот метод наследуется классом Semaphore от класса WaitHandle. Метод

WaitOne() ожидает до тех пор, пока не будет получен семафор, для которого он вызывается.

Таким образом, он блокирует выполнение вызывающего потока до тех пор,

пока указанный семафор не предоставит разрешение на доступ к ресурсу.

Если коду больше не требуется владеть семафором, он освобождает его, вызывая

метод Release(). Ниже приведены две формы этого метода.

public int Release()

public int Release(int releaseCount)

В первой форме метод Release() высвобождает только одно разрешение, а во

второй форме — количество разрешений, определяемых параметром *releaseCount.*

В обеих формах данный метод возвращает подсчитанное количество разрешений, существовавших

до высвобождения.

Метод WaitOne() допускается вызывать в потоке несколько раз перед вызовом метода

Release(). Но количество вызовов метода WaitOne() должно быть равно количеству

вызовов метода Release() перед высвобождением разрешения. С другой

стороны, можно воспользоваться формой вызова метода Release(int *num*), чтобы

передать количество высвобождаемых разрешений, равное количеству вызовов метода

WaitOne().

Ниже приведен пример программы, в которой демонстрируется применение семафора.

В этой программе семафор используется в классе MyThread для одновременного

выполнения только двух потоков типа MyThread. Следовательно, разделяемым

ресурсом в данном случае является ЦП.

(***glava23\_12***)

//this thread allows only

//two copies at a time

class MyThread

{

public Thread Thrd;

//here is created Semaphore, it gives only two

//allows from two that it has

static Semaphore sem = new Semaphore(2, 2);

public MyThread(string name)

{

Thrd = new Thread(this.Run);

Thrd.Name = name;

Thrd.Start();

}

//point of start

void Run()

{

Console.WriteLine(Thrd.Name + " waits for permit.");

sem.WaitOne();

Console.WriteLine(Thrd.Name + " got permission.");

for(char ch = 'A'; ch < 'D'; ch++)

{

Console.WriteLine(Thrd.Name + " : " + ch + " ");

Thread.Sleep(500);

}

Console.WriteLine(Thrd.Name + " released permission.");

//release Semaphor

sem.Release();

}

}

class SemaphoreDemo

{

static void Main()

{

//build 3 threads

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

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

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

mt1.Thrd.Join();

mt2.Thrd.Join();

mt3.Thrd.Join();

}

}

В классе MyThread объявляется семафор sem, как показано ниже.

static Semaphore sem = new Semaphore(2, 2);

При этом создается семафор, способный дать не более двух разрешений на доступ

к ресурсу из двух первоначально имеющихся разрешений.

Обратите внимание на то, что выполнение метода MyThread.Run() не может быть

продолжено до тех пор, пока семафор sem не даст соответствующее разрешение. Если

разрешение отсутствует, то выполнение потока приостанавливается. Когда же разрешение

появляется, выполнение потока возобновляется. В методе Main() создаются

три потока. Но выполняться могут только два первых потока, а третий должен ожидать

окончания одного из этих двух потоков.

Семафор, созданный в предыдущем примере, известен только тому процессу, который

его породил. Но семафор можно создать и таким образом, чтобы он был известен

где-нибудь еще. Для этого он должен быть именованным. Ниже приведены формы

конструктора класса Semaphore, предназначенные для создания такого семафора.

public Semaphore(int initialCount, int maximumCount, string имя)

public Semaphore(int initialCount, int maximumCount, string имя,

out bool createdNew)

В обеих формах *имя* обозначает конкретное имя, передаваемое конструктору. Если

в первой форме семафор, на который указывает *имя,* еще не существует, то он создается

с помощью значений, определяемых параметрами *initialCount* и *maximumCount.*

А если он уже существует, то значения параметров *initialCount* и *maximumCount*

игнорируются. После возврата из второй формы конструктора параметр *createdNew*

будет иметь логическое значение true, если семафор был создан. В этом случае значения

параметров *initialCount* и *maximumCount* используются для создания семафора.

Если же параметр createdNew будет иметь логическое значение false, значит, семафор

уже существует и значения параметров *initialCount* и *maximumCount* игнорируются.

Существует и третья форма конструктора класса Semaphore, в которой допускается

указывать управляющий доступом объект типа SemaphoreSecurity. С помощью

именованных семафоров можно синхронизировать взаимодействие процессов.

**Применение событий**

Для синхронизации в C# предусмотрен еще один тип объекта: событие. Существуют

две разновидности событий: устанавливаемые в исходное состояние вручную и автоматически.

Они поддерживаются в классах ManualResetEvent и AutoResetEvent

соответственно. Эти классы являются производными от класса EventWaitHandle, находящегося

на верхнем уровне иерархии классов, и применяются в тех случаях, когда

один поток ожидает появления некоторого события в другом потоке. Как только такое

событие появляется, второй поток уведомляет о нем первый поток, позволяя тем самым

возобновить его выполнение.

Ниже приведены конструкторы классов ManualResetEvent и AutoResetEvent.

public ManualResetEvent(bool initialState)

public AutoResetEvent(bool initialState)

Если в обеих формах параметр *initialState* имеет логическое значение true, то

о событии первоначально уведомляется. А если он имеет логическое значение false,

то о событии первоначально не уведомляется.

Применяются события очень просто. Так, для события типа ManualResetEvent

порядок применения следующий. Поток, ожидающий некоторое событие, вызывает

метод WaitOne() для событийного объекта, представляющего данное событие. Если

событийный объект находится в сигнальном состоянии, то происходит немедленный

возврат из метода WaitOne(). В противном случае выполнение вызывающего потока

приостанавливается до тех пор, пока не будет получено уведомление о событии. Как

только событие произойдет в другом потоке, этот поток установит событийный объект

в сигнальное состояние, вызвав метод Set(). Поэтому метод Set() следует рассматривать

как уведомляющий о том, что событие произошло. После установки событийного

объекта в сигнальное состояние произойдет немедленный возврат из метода

WaitOne(), и первый поток возобновит свое выполнение. А в результате вызова метода

Reset() событийный объект возвращается в несигнальное состояние.

Событие типа AutoResetEvent отличается от события типа ManualResetEvent

лишь способом установки в исходное состояние. Если для события типа

ManualResetEvent событийный объект остается в сигнальном состоянии до тех пор,

пока не будет вызван метод Reset(), то для события типа AutoResetEvent событийный

объект автоматически переходит в несигнальное состояние, как только поток,

ожидающий это событие, получит уведомление о нем и возобновит свое выполнение.

Поэтому если применяется событие типа AutoResetEvent, то вызывать метод

Reset() необязательно.

В приведенном ниже примере программы демонстрируется применение события

типа ManualResetEvent.

(***glava23\_13***)

//this thread notifys event passed to constructor

class MyThread

{

public Thread Thrd;

ManualResetEvent mre;

public MyThread(string name, ManualResetEvent evt)

{

Thrd = new Thread(this.Run);

Thrd.Name = name;

mre = evt;

Thrd.Start();

}

//point of start

void Run()

{

Console.WriteLine("Inside " + Thrd.Name);

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

{

Console.WriteLine(Thrd.Name);

Thread.Sleep(500);

}

Console.WriteLine(Thread.CurrentThread.Name + " done!");

//notify about event

mre.Set();

}

}

class ManualEventDemo

{

static void Main()

{

ManualResetEvent evtObj = new ManualResetEvent(false);

MyThread mt1 = new MyThread("Event Thread #1", evtObj);

Console.WriteLine("Main thread is waiting for event.");

//waiting for event

evtObj.WaitOne();

Console.WriteLine("Main thread got event from first thread.");

//set notification default state

evtObj.Reset();

mt1 = new MyThread("Event Thread #2", evtObj);

//wait for notification event

evtObj.WaitOne();

Console.WriteLine("Main thread got event from second thread.");

}

}

Прежде всего обратите внимание на то, что событие типа ManualResetEvent

передается непосредственно конструктору класса MyThread. Когда завершается метод

Run() из класса MyThread, он вызывает для событийного объекта метод Set(),

устанавливающий этот объект в сигнальное состояние. В методе Main() формируется

событийный объект evtObj типа ManualResetEvent, первоначально устанавливаемый

в исходное, несигнальное состояние. Затем создается экземпляр объекта типа

MyThread, которому передается событийный объект evtObj. После этого основной

поток ожидает уведомления о событии. А поскольку событийный объект evtObj первоначально

находится в несигнальном состоянии, то основной поток вынужден ожидать

до тех пор, пока для экземпляра объекта типа MyThread не будет вызван метод

Set(), устанавливающий событийный объект evtObj в сигнальное состояние. Это

дает возможность основному потоку возобновить свое выполнение. Затем событийный

объект устанавливается в исходное состояние, и весь процесс повторяется, но на этот

раз для второго потока. Если бы не событийный объект, то все потоки выполнялись

бы одновременно, а результаты их выполнения оказались бы окончательно запутанными.

Для того чтобы убедиться в этом, попробуйте закомментировать вызов метода

WaitOne() в методе Main().

Если бы в рассматриваемой здесь программе событийный объект типа

AutoResetEvent использовался вместо событийного объекта типа ManualResetEvent,

то вызывать метод Reset() в методе Main() не пришлось бы. Ведь в этом случае событийный

объект автоматически устанавливается в несигнальное состояние, когда поток,

ожидающий данное событие, возобновляет свое выполнение. Для опробования

этой разновидности события замените в данной программе все ссылки на объект типа

ManualResetEvent ссылками на объект типа AutoResetEvent и удалите все вызовы метода

Reset(). Видоизмененная версия программы будет работать так же, как и прежде.

**Класс Interlocked**

Еще одним классом, связанным с синхронизацией, является класс Interlocked.

Этот класс служит в качестве альтернативы другим средствам синхронизации, когда

требуется только изменить значение общей переменной. Методы, доступные в классе

Interlocked, гарантируют, что их действие будет выполняться как единая, непрерываемая

операция. Это означает, что никакой синхронизации в данном случае вообще

не требуется. В классе Interlocked предоставляются статические методы для сложения

двух целых значений, инкрементирования и декрементирования целого значения,

сравнения и установки значений объекта, обмена объектами и получения 64-разрядно-

го значения. Все эти операции выполняются без прерывания.

В приведенном ниже примере программы демонстрируется применение двух методов

из класса Interlocked:Increment() и Decrement(). При этом используются

следующие формы обоих методов:

public static int Increment(ref int location)

public static int Decrement(ref int location)

где *location —* это переменная, которая подлежит инкрементированию или декрементированию.

(***glava23\_14***)

//common resourse

class SharedRes

{

public static int Count = 0;

}

//this thread variable SharedRes.Count inrements

class IncThread

{

public Thread Thrd;

public IncThread(string name)

{

Thrd = new Thread(this.Run);

Thrd.Name = name;

Thrd.Start();

}

//point of start

void Run()

{

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

{

Interlocked.Increment(ref SharedRes.Count);

Console.WriteLine(Thrd.Name + " Count = " + SharedRes.Count);

}

}

}

//this thread Shared.Count decrements

class DecThread

{

public Thread Thrd;

public DecThread(string name)

{

Thrd = new Thread(this.Run);

Thrd.Name = name;

Thrd.Start();

}

//point of start

void Run()

{

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

{

Interlocked.Decrement(ref SharedRes.Count);

Console.WriteLine(Thrd.Name + " Count = " + SharedRes.Count);

}

}

}

class InterLockedDemo

{

static void Main()

{

//build two threads

IncThread mt1 = new IncThread("Increment Thread");

DecThread mt2 = new DecThread("Decrement Thread");

mt1.Thrd.Join();

mt2.Thrd.Join();

}

}

**Классы синхронизации, внедренные в версии .NET Framework 4.0**

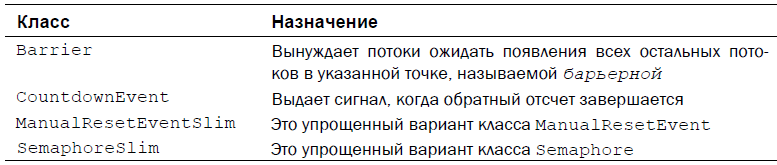
Рассматривавшиеся ранее классы синхронизации, в том числе Semaphore и

AutoResetEvent, были доступны в среде .NET Framework, начиная с версии 1.1.

Таким образом, эти классы образуют основу поддержки синхронизации в среде .NET

Framework. Но после выпуска версии .NET Framework 4.0 появился ряд новых альтернатив

этим классам синхронизации. Все они перечисляются ниже.



Если вам понятно, как пользоваться основными классами синхронизации, описанными

ранее в этой главе, то у вас не должно возникнуть затруднений при использовании

их новых альтернатив и дополнений.

**Прерывание потока**

Иногда поток полезно прервать до его нормального завершения. Например, отладчику

может понадобиться прервать вышедший из-под контроля поток. После прерывания

поток удаляется из системы и не может быть начат снова.

Для прерывания потока до его нормального завершения служит метод Thread.

Abort(). Ниже приведена простейшая форма этого метода.

public void Abort()

Метод Abort() создает необходимые условия Для генерирования исключения

ThreadAbortException в том потоке, для которого он был вызван. Это исключение

приводит к прерыванию потока и может быть перехвачено и в коде программы, но

в этом случае оно автоматически генерируется еще раз, чтобы остановить поток. Метод

Abort() не всегда способен остановить поток немедленно, поэтому если поток

требуется остановить перед тем, как продолжить выполнение программы, то после

метода Abort() следует сразу же вызвать метод Join(). Кроме того, в самых редких

случаях методу Abort() вообще не удается остановить поток. Это происходит, например,

в том случае, если кодовый блок finally входит в бесконечный цикл.

В приведенном ниже примере программы демонстрируется применение метода

Abort() для прерывания потока.

(***glava23\_15***)

class MyThread

{

public Thread Thrd;

public MyThread(string name)

{

Thrd = new Thread(this.Run);

Thrd.Name = name;

Thrd.Start();

}

//point of start

void Run()

{

Console.WriteLine(Thrd.Name + " started.");

for(int i = 0; i <= 1000; i++)

{

Console.Write(i + " ");

if((i%10)==0)

{

Console.WriteLine();

Thread.Sleep(250);

}

}

Console.WriteLine(Thrd.Name + " done.");

}

}

class StopDemo

{

static void Main()

{

MyThread mt1 = new MyThread("My Thread");

Thread.Sleep(1000);//allow derived thread to start

Console.WriteLine("Interupt the Thread.");

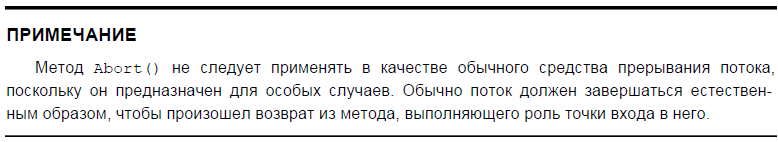
mt1.Thrd.Abort();

mt1.Thrd.Join(); //wait for interuption

Console.WriteLine("Main thread interupted.");

}

}



**Другая форма метода Abort()**

В некоторых случаях оказывается полезной другая форма метода Abort(), приведенная

ниже в общем виде:

public void Abort(object stateInfo)

где *stateInfo* обозначает любую информацию, которую требуется передать потоку,

когда он останавливается. Эта информация доступна посредством свойства

ExceptionState из класса исключения ThreadAbortException. Подобным образом

потоку можно передать код завершения. В приведенном ниже примере программы

демонстрируется применение данной формы метода Abort().

(***glava23\_16***)

class MyThread

{

public Thread Thrd;

public MyThread(string name)

{

Thrd = new Thread(this.Run);

Thrd.Name = name;

Thrd.Start();

}

void Run()

{

try

{

Console.WriteLine(Thrd.Name + " started.");

for (int i = 1; i <= 1000; i++)

{

Console.Write(i + " ");

if ((i % 10) == 0)

{

Console.WriteLine();

Thread.Sleep(250);

}

}

Console.WriteLine(Thrd.Name + " ended normaly.");

}

catch (ThreadAbortException exc)

{

Console.WriteLine("Thread Aborted, code of ending " + exc.ExceptionState);

}

}

}

class UseAltAbort

{

static void Main()

{

MyThread mt1 = new MyThread("My Thread");

Thread.Sleep(1000);//wait few seconds before abort thread

Console.WriteLine("Aborting Thread.");

mt1.Thrd.Abort(100);

mt1.Thrd.Join();

Console.WriteLine("Main thread aborted.");

}

}

Как следует из приведенного выше результата, значение 100 передается методу

Abort() в качестве кода прерывания. Это значение становится затем доступным посредством

свойства ExceptionState из класса исключения ThreadAbortException,

которое перехватывается потоком при его прерывании.

**Отмена действия метода Abort()**

Запрос на преждевременное прерывание может быть переопределен в самом

потоке. Для этого необходимо сначала перехватить в потоке исключение

ThreadAbortException, а затем вызвать метод ResetAbort(). Благодаря этому

исключается повторное генерирование исключения по завершении обработчика исключения,

прерывающего данный поток. Ниже приведена форма объявления метода

ResetAbort().

public static void ResetAbort()

Вызов метода ResetAbort() может завершиться неудачно, если в потоке отсутствует

надлежащий режим надежной отмены преждевременного прерывания потока.

В приведенном ниже примере программы демонстрируется применение метода

ResetAbort().

(***glava23\_17***)

class MyThread

{

public Thread Thrd;

public MyThread(string name)

{

Thrd = new Thread(this.Run);

Thrd.Name = name;

Thrd.Start();

}

void Run()

{

Console.WriteLine(Thrd.Name + " started.");

for (int i = 1; i <= 1000; i++)

{

try

{

Console.Write(i + " ");

if ((i % 10) == 0)

{

Console.WriteLine();

Thread.Sleep(250);

}

}

catch (ThreadAbortException exc)

{

if ((int)exc.ExceptionState == 0)

{

Console.WriteLine("Abortion of thread canceled! "

+ "Code of ending " + exc.ExceptionState);

Thread.ResetAbort();

}

else

Console.WriteLine("Thread Aborted, code of ending " + exc.ExceptionState);

}

}

Console.WriteLine(Thrd.Name + " ended normaly.");

}

}

class ResetAbort

{

static void Main()

{

MyThread mt1 = new MyThread("My thread");

Thread.Sleep(1000);

Console.WriteLine("Abroting thread.");

mt1.Thrd.Abort(0);//this wount stop thread

Thread.Sleep(1000);

Console.WriteLine("Aborting thread again.");

mt1.Thrd.Abort(100);//this will abort

mt1.Thrd.Join();

Console.WriteLine("Main thread interupted.");

}

}

Если в данном примере программы метод Abort() вызывается с нулевым аргументом,

то запрос на преждевременное прерывание отменяется потоком, вызывающим

метод ResetAbort(), и выполнение этого потока продолжается. Любое другое значение

аргумента приведет к прерыванию потока.

**Приостановка и возобновление потока**

В первоначальных версиях среды .NET Framework поток можно было приостановить

вызовом метода Thread.Suspend() и возобновить вызовом метода Thread.

Resume(). Но теперь оба эти метода считаются устаревшими и не рекомендуются к

применению в новом коде. Объясняется это, в частности, тем, что пользоваться методом

Suspend() на самом деле небезопасно, так как с его помощью можно приостановить

поток, который в настоящий момент удерживает блокировку, что препятствует

ее снятию, а следовательно, приводит к взаимоблокировке. Применение обоих методов

может стать причиной серьезных осложнений на уровне системы. Поэтому для

приостановки и возобновления потока следует использовать другие средства синхронизации,

в том числе мьютекс и семафор.

**Определение состояния потока**

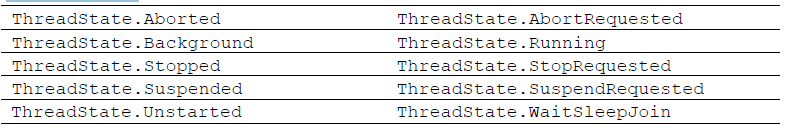
Состояние потока может быть получено из свойства ThreadState, доступного в

классе Thread. Ниже приведена общая форма этого свойства.

public ThreadState ThreadState { get; }

Состояние потока возвращается в виде значения, определенного в перечислении

ThreadState. Ниже приведены значения, определенные в этом перечислении.



Все эти значения не требуют особых пояснений, за исключением одного. Значение

ThreadState.WaitSleepJoin обозначает состояние, в которое поток переходит во

время ожидания в связи с вызовом метода Wait(), Sleep() или Join().

**Применение основного потока**

Как пояснялось в самом начале этой главы, у всякой программы на C# имеется хотя

бы один поток исполнения, называемый *основным.* Этот поток программа получает

автоматически, как только начинает выполняться. С основным потоком можно обращаться

таким же образом, как и со всеми остальными потоками.

Для доступа к основному потоку необходимо получить объект типа Thread, который

ссылается на него. Это делается с помощью свойства CurrentThread, являющегося

членом класса Thread. Ниже приведена общая форма этого свойства.

public static Thread CurrentThread { get; }

Данное свойство возвращает ссылку на тот поток, в котором оно используется. Поэтому

если свойство CurrentThread используется при выполнении кода в основном

потоке, то с его помощью можно получить ссылку на основной поток. Имея в своем

распоряжении такую ссылку, можно управлять основным потоком так же, как и любым

другим потоком.

В приведенном ниже примере программы сначала получается ссылка на основной

поток, а затем получаются и устанавливаются имя и приоритет основного потока.

(***glava23\_18***)

class UseMain

{

static void Main()

{

Thread Thrd;

//get main thread

Thrd = Thread.CurrentThread;

//show name of it

if (Thrd.Name == null)

Console.WriteLine("Main thread has no name.");

else

Console.WriteLine("Name of main thread: " + Thrd.Name);

//show priority main thread

Console.WriteLine("Priority: " + Thrd.Priority);

Console.WriteLine();

//set name and priority

Console.WriteLine("Set name and priority.");

Thrd.Name = "Main thread";

Thrd.Priority = ThreadPriority.AboveNormal;

Console.WriteLine("Now main thread is: " + Thrd.Name);

Console.WriteLine("And priority is: " + Thrd.Priority);

}

}

Следует, однако, быть очень внимательным, выполняя операции с основным потоком.

Так, если добавить в конце метода Main() следующий вызов метода Join():

Thrd.Join();

программа никогда не завершится, поскольку она будет ожидать окончания основного

потока!

**Дополнительные средства многопоточной обработки,**

**внедренные в версии .NET Framework 4.0**

В версии .NET Framework 4.0 внедрен ряд новых средств многопоточной обработки,

которые могут оказаться весьма полезными. Самым важным среди них является новая

система отмены. В этой системе поддерживается механизм отмены потока простым,

вполне определенным и структурированным способом. В основу этого механизма положено

понятие *признака отмены, с* помощью которого указывается состояние отмены

потока. Признаки отмены поддерживаются в классе CancellationTokenSource и в

структуре CancellationToken. Система отмены полностью интегрирована в новую

библиотеку распараллеливания задач (TPL), и поэтому она подробнее рассматривается

вместе с TPL в главе 24.

В класс System.Threading добавлена структура SpinWait, предоставляющая методы

SpinOnce() и SpinUntil(), которые обеспечивают более полный контроль над

ожиданием в состоянии занятости. Вообще говоря, структура SpinWait оказывается

непригодной для однопроцессорных систем. А для многопроцессорных систем она

применяется в цикле. Еще одним элементом, связанным с ожиданием в состоянии занятости,

является структура SpinLock, которая применяется в цикле ожидания до тех

пор, пока не станет доступной блокировка. В класс Thread добавлен метод Yield(),

который просто выдает остаток кванта времени, выделенного потоку. Ниже приведена

общая форма объявления этого метода.

public static bool Yield()

Этот метод возвращает логическое значение true, если происходит переключение

контекста. В отсутствие другого потока, готового для выполнения, переключение контекста

не произойдет.

**Рекомендации по многопоточному программированию**

Для эффективного многопоточного программирования самое главное — мыслить

категориями параллельного, а не последовательного выполнения кода. Так, если в

одной программе имеются две подсистемы, которые могут работать параллельно, их

следует организовать в отдельные потоки. Но делать это следует очень внимательно

и аккуратно, поскольку если создать слишком много потоков, то тем самым можно

значительно снизить, а не повысить производительность программы. Следует также

иметь в виду дополнительные издержки, связанные с переключением контекста. Так,

если создать слишком много потоков, то на смену контекста уйдет больше времени

ЦП, чем на выполнение самой программы! И наконец, для написания нового кода,

предназначенного для многопоточной обработки, рекомендуется пользоваться библиотекой

распараллеливания задач (TPL), о которой речь пойдет в следующей главе.

**Запуск отдельной задачи**

Многозадачность на основе потоков чаще всего организуется при программировании

на С#. Но там, где это уместно, можно организовать и многозадачность на основе

процессов. В этом случае вместо запуска другого потока в одной и той же программе

одна программа начинает выполнение другой. При программировании на C# это

делается с помощью класса Process, определенного в пространстве имен System.

Diagnostics. В заключение этой главы вкратце будут рассмотрены особенности запуска

и управления другим процессом.

Простейший способ запустить другой процесс — воспользоваться методом

Start(), определенным в классе Process. Ниже приведена одна из самых простых

форм этого метода:

public static Process Start(string имя\_файла)

где *имя\_файла* обозначает конкретное имя файла, который должен исполняться или

же связан с исполняемым файлом.

Когда созданный процесс завершается, следует вызвать метод Close(), чтобы освободить

память, выделенную для этого процесса. Ниже приведена форма объявления

метода Close().

public void Close()

Процесс может быть прерван двумя способами. Если процесс является приложением

Windows с графическим пользовательским интерфейсом, то для прерывания

такого процесса вызывается метод CloseMainWindow(), форма которого приведена

ниже.

public bool CloseMainWindow()

Этот метод посылает процессу сообщение, предписывающее ему остановиться. Он

возвращает логическое значение true, если сообщение получено, и логическое значение

false, если приложение не имеет графического пользовательского интерфейса

или главного окна. Следует, однако, иметь в виду, что метод CloseMainWindow() служит

только для запроса остановки процесса. Если приложение проигнорирует такой

запрос, то оно не будет прервано как процесс.

Для безусловного прерывания процесса следует вызвать метод Kill(), как показано

ниже.

public void Kill()

Но методом Kill() следует пользоваться аккуратно, так как он приводит к неконтролируемому

прерыванию процесса. Любые несохраненные данные, связанные с прерываемым

процессом, будут, скорее всего, потеряны.

Для того чтобы организовать ожидание завершения процесса, можно воспользоваться

методом WaitForExit(). Ниже приведены две его формы.

public void WaitForExit()

public bool WaitForExit(int миллисекунд)

В первой форме ожидание продолжается до тех пор, пока процесс не завершится,

а во второй форме — только в течение указанного количества *миллисекунд.* В последнем

случае метод WaitForExit() возвращает логическое значение true, если процесс

завершился, и логическое значение false, если он все еще выполняется.

В приведенном ниже примере программы демонстрируется создание, ожидание и

закрытие процесса. В этой программе сначала запускается стандартная сервисная программа

Windows: текстовый редактор WordPad.exe, а затем организуется ожидание

завершения программы WordPad как процесса.

(***glava23\_19***)

using System.Diagnostics;

class StartProcess

{

static void Main()

{

Process newProc = Process.Start("wordpad.exe");

Console.WriteLine("New proccess started.");

newProc.WaitForExit();

newProc.Close();//clean resourses

Console.WriteLine("New process stoped.");

}

}

При выполнении этой программы запускается стандартное приложение WordPad,

и на экране появляется сообщение "Новый процесс запущен.". Затем программа

ожидает закрытия WordPad. По окончании работы WordPad на экране появляется заключительное

сообщение "Новый процесс завершен.".