# 14 Лабораторная работа № 11

**Тема:** Многопоточное программирование.

**Цель:** изучение средств языка С# по созданию и управлению потоками в многопоточных программах

## **14.1 Содержание работы**

1. Ознакомиться с теоретическим материалом, приведённом в разделе 14.2 данной лабораторной работы.
2. Выполнить задание согласно своему варианту. Перечень заданий приведён в разделе 14.3 данной лабораторной работы.
3. Оформить отчёт о выполнении работы.
4. Ответить на контрольные вопросы, приведённые в разделе 14.5.

## **14.2 Теоретический материал**

Среди множества новых средств С# самым значительным, пожалуй, является встроенная поддержка многопоточного программирования (multithreaded programming).

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

Таким образом, многопоточность представляет собой специальную форму многозадачности. Многопоточное программирование опирается на сочетание средств, предусмотренных языком С#, и классов, определенных в среде .NET Framework.

Многие проблемы, связанные с многопоточностью, которые имеют место в других языках, в С# минимизированы или устранены совсем, поскольку поддержка многопоточности встроена в язык.

**Основы многопоточности**

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

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

Итак, различие между процессно - и поточно-ориентированной многозадачностями

можно определить следующим образом. Процессно-ориентированная многозадачность обеспечивает одновременное выполнение программ, а поточно-ориентированная - одновременное выполнение частей одной и той же программы. Преимущество многопоточности состоит в том, что она позволяет писать очень эффективные программы, поскольку предоставляет возможность с толком использовать вынужденное время ожидания (простоя), которое имеет место во многих программах. Общеизвестно, что большинство устройств ввода-вывода (сетевые порты, дисководы или клавиатура) работают гораздо медленнее, чем центральный процессор (CPU). Поэтому в программах львиная доля времени выполнения зачастую тратится на ожидание окончания отправки информации устройству (или получения от него). Используя многопоточность, можно построить программу так, чтобы она в такие периоды ожидания выполняла другую задачу. Например, пока одна часть программы будет отправлять файл по электронной почте, другая ее часть может считывать входные данные с клавиатуры, а еще одна - буферизировать следующий блок данных для отправки в Internet.

Поток может находиться в одном из нескольких возможных состояний. Он может выполняться. Он может быть готовым к выполнению (как только получит время CPU). Выполняющийся поток может быть приостановлен, т.е. его выполнение временно прекращается. Позже оно может быть возобновлено. Поток может быть заблокирован в ожидании необходимого ресурса. Наконец, поток может завершиться, и уж в этом случае его выполнение окончено и продолжению (возобновлению) не подлежит. В среде .NET Framework определено два типа потоков: высокоприоритетный (foreground) и низкоприоритетный, или фоновый (background). По умолчанию поток создается высокоприоритетным, но его тип можно изменить, т.е. сделать его фоновым. Единственное различие между высоко - и низкоприоритетным потоками состоит в том, что последний будет автоматически завершен, если все высокоприоритетные потоки в его процессе остановились. Поточно-ориентированная многозадачность не может обойтись без специального средства, именуемого синхронизацией, которое позволяет координировать выполнение потоков вполне определенными способами. В С# предусмотрена отдельная подсистема, посвященная синхронизации, ключевые средства которой здесь также рассматриваются. Все процессы имеют по крайней мере один поток управления, который обычно называется основным (main thread), поскольку именно с этого потока начинается выполнение программы. Таким образом, все программы имеют по крайней мере один основной поток. Из основного можно создать и другие потоки.

Язык С# и среда .NET Framework поддерживают как процессно -, так и поточно-ориентированную многозадачность. Следовательно, используя С#, можно создавать как процессы, так и потоки, а затем ими эффективно управлять. При этом, чтобы создать новый процесс, потребуется написать небольшую программку, поскольку каждый процесс в значительной степени отделен от следующего. Здесь важно то, что С# обеспечивает поддержку многопоточности. Поскольку поддержка многопоточности является встроенной, С# значительно упрощает создание высокоэффективных многопоточных программ по сравнению с другими языками, например по сравнению с C++ (в который не встроена поддержка многопоточности).

Классы, которые поддерживают многопоточное программирование, определены в

пространстве имен System.Threading. Поэтому в начало любой многопоточной программы необходимо включить следующую инструкцию:

using System.Threading;

**Класс Thread**

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

**Создание потока**

Чтобы создать поток, необходимо создать объект типа Thread. В классе Thread определен следующий конструктор:

public Thread (ThreadStart *entryPoint*)

Здесь параметр *entryPoint* содержит имя метода, который будет вызван, чтобы начать выполнение потока. Тип ThreadStart – это делегат, определенный в среде .NET Framework:

public delegate void ThreadStart()

Итак, начальный метод должен иметь тип возвращаемого значения void и не принимать никаких аргументов.

Выполнение созданного потока не начнется до тех пор, пока не будет вызван метод Start(), который определяется в классе Thread. Его определение выглядит так:

public void Start()

Начавшись, выполнение потока будет продолжаться до тех пор, пока не завершится метод, заданный параметром *entryPoint*. Поэтому после выхода из *entryPoint* – метода выполнение потока автоматически завершается. Если попытаться вызвать метод Start() для потока, запущенного на выполнение, будет сгенерировано исключения типа ThreadStateException.

Рассмотрим пример создания нового потока и начала его выполнения (Пример 1):

using System;

using System.Threading; //необходимо подключать это пространство

//для многопоточных приложений

class MyThread

{

public int count;//объявление членов экземпляра класса

string thrdName;

public MyThread(string name)

{

count = 0;

thrdName = name;

}

//Начало (входная точка) потока

public void run()

{

Console.WriteLine(thrdName + "стартовал.");

do

{

Thread.Sleep(500);

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

count++;

} while (count < 10);

Console.WriteLine(thrdName + " завершен.");

}

}

class MultiThread

{

public static void Main()

{

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

//Создаем объект класса MyThread

MyThread mt = new MyThread("Потомок №1");

//Затем из этого объекта создаем поток

Thread newThrd = new Thread(new ThreadStart(mt.run));

//Запускаем выполнение потока

newThrd.Start();

do

{

Console.Write(".");

Thread.Sleep(100);

} while (mt.count != 10);

Console.WriteLine("Основной поток завершен");

Console.ReadKey();

}

}

Рассмотрим внимательно эту программу. Класс MyThread используется для создания второго потока управления. В его методе run() организован цикл, который при каждой итерации увеличивает значение переменной count на единицу, пока значение в переменной count<10. Обратите внимание на вызов метода Sleep(), который является статическим и определен в классе Thread. Метод Sleep() заставляет поток, из которого он был вызван, приостановить выполнение на период времени, заданный в миллисекундах. В нашей программе проиллюстрирован следующий формат использования этого метода:

public static void Sleep(int milliseconds)

В параметре milliseconds задается время в миллисекундах, на которое будет приостановлено выполнение потока. Если параметр milliseconds равен нулю, вызывающий поток приостанавливается только для того, чтобы ожидающему потоку разрешить выполнение.

В методе Main() при выполнении следующих инструкций создается новый объект класса Thread:

//Создаем объект класса MyThread

MyThread mt = new MyThread("Потомок №1");

//Затем из этого объекта создаем поток

Thread newThrd = new Thread(new ThreadStart(mt.run));

//Запускаем выполнение потока

newThrd.Start();

Как подсказывают комментарии, сначала создается объект класса MyThread. Этот объект затем используется для создания объекта класса Thread путем передачи значения mt.run в качестве имени стартового метода, или метода входной точки. Наконец, вызов метода Start() запускает новый поток на выполнение. Это приводит к вызову метода run() дочернего потока. После вызова метода Start() выполнение основного потока возвращается в метод Main(), а именно в цикл do. Теперь выполняются оба потока, разделяя время CPU до тех пор, пока не закончатся их циклы [3]. Программа генерирует следующие результаты (рисунок 19):

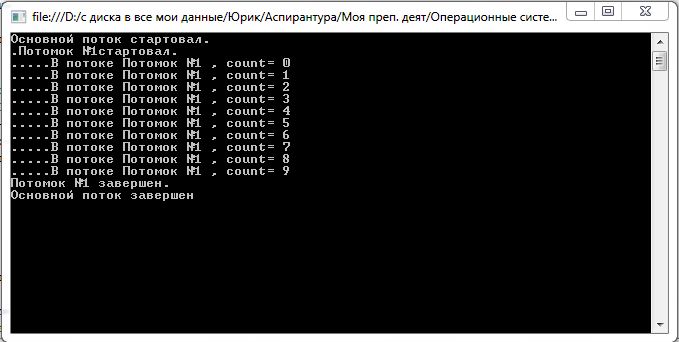


Рисунок 19 – Результаты работы программы из примера

Часто в многопоточной программе нужно позаботиться о том, чтобы основной поток завершался последним. Формально программа продолжает выполняться до тех пор, пока не завершатся все высокоприоритетные потоки. Таким образом, совсем не обязательно завершение основного потока последним. Однако добиваться этого считается одним из признаков хорошего стиля программирования, поскольку в этом случае ясно определяется конечная точка программы. В предыдущем примере основной поток гарантированно завершается последним, поскольку цикл do останавливается, когда значение переменной count становится равным 10. Поскольку переменная count примет значение 10 только после того, как завершится выполнение потокового объекта newThrd, основной поток закончится последним. Далее будут показаны более удачные способы ожидания одним потоком завершения другого. Несмотря на то что предыдущая программа вполне работоспособна, несколько простых усовершенствований сделают ее более эффективной. Во-первых, можно организовать начало выполнения потока сразу после его создания. Для потока класса MyThread это достижимо посредством создания объекта типа Thread внутри конструктора класса MyThread. Во-вторых, не обязательно хранить имя потока в классе MyThread, поскольку в классе Thread определено свойство Name, которое можно использовать с этой целью. Свойство Name определено таким образом:

public string Name { get; set ; }

Так как свойство Name предназначено для чтения и записи, его можно использовать для запоминания имени потока или для его считывания.

Вот как выглядит усовершенствованная версия предыдущей программы (Пример 2):

using System;

using System.Threading; //необходимо подключать это пространство

//для многопоточных приложений

class MyThread

{

public int count;

public Thread thrd;

//Конструктор класса MyThread

public MyThread(string name)

{

count = 0;

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

thrd.Name = name; //Устанавливаем имя потока

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

}

//Входная точка потока (метод run)

void run()

{

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

do

{

Thread.Sleep(500);

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

count++;

} while (count < 10);

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

}

}

class MultiThread

{

public static void Main()

{

Console.WriteLine("Основной поток стартовал");

//Создаем объект класса MyThread

MyThread mt = new MyThread("Потомок №1");

do

{

Console.Write(".");

Thread.Sleep(100);

} while (mt.count != 10);

Console.WriteLine("Основной поток завершен");

Console.ReadKey();

}

}

Эта версия программы генерирует те же результаты, что и предыдущая. Обратите внимание на то, что потоковый объект хранится теперь в переменной thrd внутри класса MyThread [3].

**Создание нескольких потоков**

В предыдущих примерах создавался только один дочерний поток. Однако программа способна порождать столько потоков, сколько потребуется в конкретной ситуации. Например, следующая программа создает три дочерних потока (Пример 3):

using System;

using System.Threading; //необходимо подключать это пространство

//для многопоточных приложений

class MyThread

{

public int count;

public Thread thrd;

//Конструктор класса MyThread

public MyThread(string name)

{

count = 0;

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

thrd.Name = name; //Устанавливаем имя потока

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

}

//Входная точка потока (метод run)

void run()

{

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

do

{

Thread.Sleep(500);

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

count++;

} while (count < 10);

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

}

}

class MoreThread

{

public static void Main()

{

Console.WriteLine("Основной поток стартовал");

//Создаем три потока

MyThread mt1 = new MyThread("Потомок №1");

MyThread mt2 = new MyThread("Потомок №2");

MyThread mt3 = new MyThread("Потомок №3");

do

{

Console.Write(".");

Thread.Sleep(100);

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

Console.WriteLine("Основной поток завершен");

Console.ReadKey();

}

}

На рисунке 20 показано окно с возможными результатами программы. Как можно судить по приведенным результатам, сразу после старта все три потока разделяют время CPU. Из-за различий в системных конфигурациях, операционных системах и других факторах среды результаты выполнения этой программы на разных компьютерах могут незначительно отличаться от представленных здесь.

Иногда полезно знать, когда завершится выполнение потока. В предыдущих примерах это достигалось за счет проверки значения переменной count, но такое решение вряд ли можно считать удовлетворительным. К счастью, в классе Thread предусмотрено два средства, которые позволяют установить факт завершения выполнения потока. Одно из них - предназначенное только для чтения свойство isAlive. Оно определяется так:

public bool IsAlive { get; }

Свойство IsAlive возвращает значение true, если поток, для которого оно опрашивается, еще выполняется. В противном случае оно возвращает значение false. Чтобы опробовать эту возможность, замените следующей версией класса MoreThreads ту, что представлена в предыдущей программе (Пример 3):

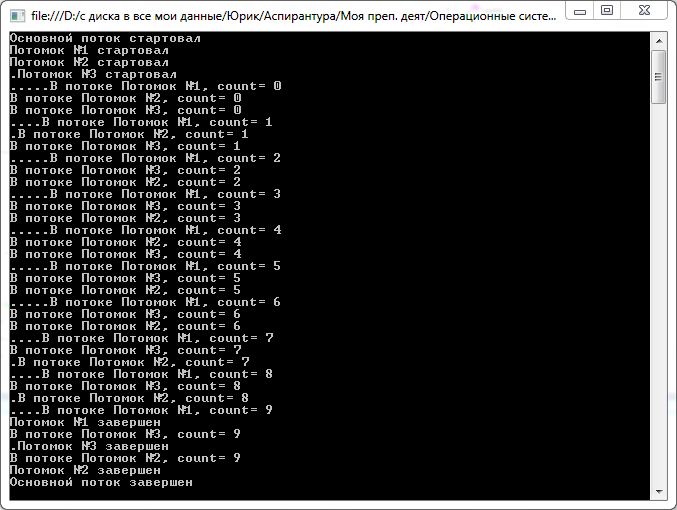


Рисунок 20 – Окно с результатами программы

//Используем свойство IsAlive для установления факта

//завершения выполнения потока

class MoreThread

{

public static void Main()

{

Console.WriteLine("Основной поток стартовал");

//Создаем три потока

MyThread mt1 = new MyThread("Потомок №1");

MyThread mt2 = new MyThread("Потомок №2");

MyThread mt3 = new MyThread("Потомок №3");

do

{

Console.Write(".");

Thread.Sleep(100);

} while (mt1.thrd.IsAlive && mt2.thrd.IsAlive && mt3.thrd.IsAlive);

Console.WriteLine("Основной поток завершен");

Console.ReadKey();

}

}

С использованием этой версии класса MoreThreads результаты работы программы

аналогичны предыдущим. Единственное различие между этими версиями состоит в том, что в последней для установления факта завершения выполнения дочерних потоков используется свойство IsAlive.

Второй способ, который позволяет "дождаться" завершения выполнения потока, состоит в вызове метода Join(). Самый простой формат его использования имеет такой вид:

public void Join()

Метод Join() ожидает, пока поток, для которого он был вызван, не завершится. Имя этого метода ("join" в перев. с англ.- присоединяться) связано с идеей вызвать указанный поток и подождать, когда тот присоединится к вызвавшему его методу. Если заданный поток даже не стартовал, будет сгенерировано исключение типа ThreadStateException. Другие форматы метода Join() позволяют указать максимальный объем времени, в течение которого вы собираетесь ожидать завершения выполнения потока.

Рассмотрим программу, которая для получения гарантии того, что основной поток завершится последним, использует метод Join() (Пример 4):

using System;

using System.Threading; //необходимо подключать это пространство

//для многопоточных приложений

class MyThread

{

public int count;

public Thread thrd;

//Конструктор класса MyThread

public MyThread(string name)

{

count = 0;

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

thrd.Name = name; //Устанавливаем имя потока

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

}

//Входная точка потока (метод run)

void run()

{

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

do

{

Thread.Sleep(500);

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

count++;

} while (count < 10);

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

}

}

//Используем метод Join() для ожидания завершения потоков

class JoinThread

{

public static void Main()

{

Console.WriteLine("Основной поток стартовал");

//Создаем три потока

MyThread mt1 = new MyThread("Потомок №1");

MyThread mt2 = new MyThread("Потомок №2");

MyThread mt3 = new MyThread("Потомок №3");

mt1.thrd.Join();

Console.WriteLine("Потомок 1 присоединен");

mt2.thrd.Join();

Console.WriteLine("Потомок 2 присоединен");

mt3.thrd.Join();

Console.WriteLine("Потомок 3 присоединен");

Console.WriteLine("Основной поток завершен");

Console.ReadKey();

}

}

Результаты работы программы приведены на рисунке 21. В результате после выхода из метода Join() выполнения потоков завершено.

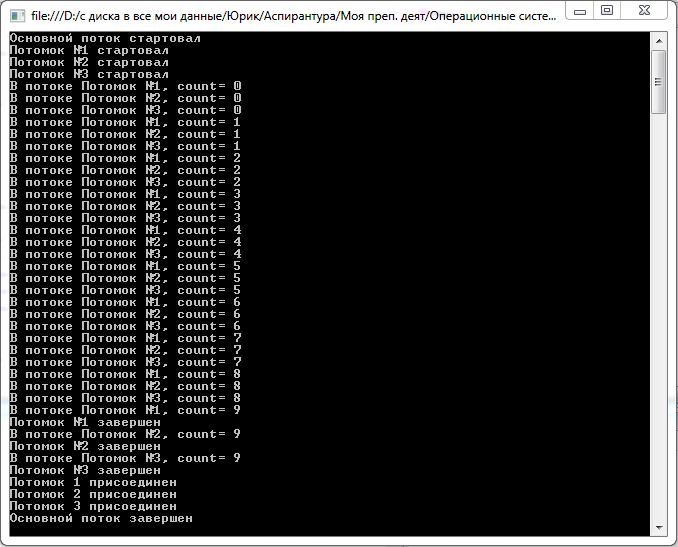


Рисунок 21 – Окно с результатами программы

**Свойство IsBackground**

Cреда .NET Framework определяет два типа потоков: высокоприоритетные и фоновые. Единственное различие между ними состоит в том, что процесс не завершится до тех пор, пока не завершится выполнение всех его высокоприоритетных потоков, при этом фоновые потоки заканчиваются автоматически после завершения всех высокоприоритетных потоков. По умолчанию любой поток создается как высокоприоритетный [3]. При необходимости его можно сделать фоновым с помощью свойства IsBackground, которое определено в классе Thread следующим образом:

public bool IsBackground { get; set; }

Чтобы перевести поток в категорию фоновых, достаточно присвоить свойству IsBackground значение true. Значение false означает, что соответствующий поток является высокоприоритетным.

**Приоритеты потоков**

Каждый поток имеет определенный приоритет. Приоритет потока, в частности, определяет, какой объем процессорного времени получает поток. В общем случае низкоприоритетным потокам выделяется немного времени CPU, а высокоприоритетным - побольше. Объем времени CPU, выделяемый потоку, в значительной степени влияет на его выполнение и взаимодействие с другими потоками, выполняющимися в данный момент в системе. Важно понимать, что существуют и другие факторы (помимо приоритета потока), которые влияют на объем времени CPU, выделяемый потоку. Например, если высокоприоритетный поток находится в состоянии ожидания некоторого ресурса, например вводимых с клавиатуры данных, он будет заблокирован, и в это время сможет работать поток с более низким приоритетом. Поэтому в подобной ситуации низкоприоритетный поток может получить более продолжительный доступ к CPU, чем высокоприоритетный. После старта дочерний поток получает стандартное значение приоритета. Его можно изменить с помощью свойства Priority, которое является членом класса Thread [4].

Общий формат его использования таков:

public ThreadPriority Priority{ get; set; }

Здесь ThreadPriority - перечисление, которое определяет следующие пять значений приоритета:

ThreadPriority.Highest

ThreadPriority.AboveNormal

ThreadPriority.Normal

ThreadPriority.BelowNoritial

ThreadPriority.Lowest

По умолчанию потоку присваивается значение приоритета ThreadPriority.Normal.

Чтобы понять, как приоритеты влияют на выполнение потоков, воспользуемся примером, в котором выполняются два потока, причем с разными приоритетами. Эти потоки создаются как экземпляры класса MyThread. Метод run() содержит цикл, в котором подсчитывается количество итераций. Цикл останавливается либо на счете 1 000 000 000, либо в тот момент, когда статическая переменная stop примет значение true. До входа в этот цикл переменная stop установлена равной false. Первый поток, "досчитав" до 1 000 000 000, устанавливает переменную stop равной значению true. Это заставляет второй поток отказаться от следующего кванта времени (объем процессорного времени, выделяемого потоку). На каждом проходе цикла содержимое строковой переменной currentName сравнивается с именем выполняющегося потока. Если они не совпадают, значит, имело место переключение задач. При каждом переключении задач отображается имя нового потока, а переменной currentName присваивается имя текущего потока. Это позволяет наблюдать за тем, как часто каждый из потоков получает доступ к CPU. По завершении выполнения обоих потоков отображается количество итераций, выполненных в каждом цикле (Пример 5).

//Демонстрация использования приоритетов потоков

using System;

using System.Threading; //необходимо подключать это пространство

//для многопоточных приложений

class MyThread

{

public int count;

public Thread thrd;

static bool stop = false;

static string currentName;

/\* Создаем новый поток. Обратите внимание на то, что этот конструктор

в действительности не запускает потоки на выполнение \*/

//Конструктор класса MyThread

public MyThread(string name)

{

count = 0;

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

thrd.Name = name; //Устанавливаем имя потока

currentName = name;

}

//Входная точка потока (метод run)

void run()

{

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

do

{

count++;

if (currentName != thrd.Name)

{

currentName = thrd.Name;

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

}

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

stop = true;

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

}

}

class PriorityDemo

{

public static void Main()

{

//Создаем два потока

MyThread mt1 = new MyThread("с высоким приоритетом");

MyThread mt2 = new MyThread("с низким приоритетом");

//Устанавливаем приоритеты

mt1.thrd.Priority = ThreadPriority.AboveNormal;

mt2.thrd.Priority = ThreadPriority.BelowNormal;

//Запускаем потоки на выполнение

mt1.thrd.Start();

mt2.thrd.Start();

mt1.thrd.Join();

mt2.thrd.Join();

Console.WriteLine();

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

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

Console.ReadKey();

}

}

Результаты работы программы приведены на рисунке 22.

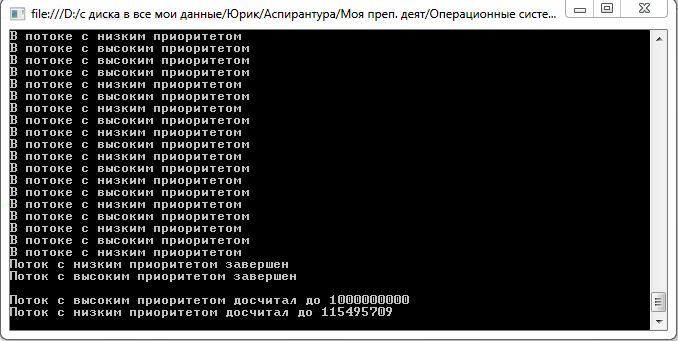


Рисунок 22 – Окно с результатами программы

Судя по полученным результатам, поток с высоким приоритетом получил приблизительно 89% процессорного времени. Конечно же, результаты, полученные на другом компьютере, могут отличаться от приведенных, поскольку они зависят от быстродействия CPU и количества их задач, выполняемых в системе. Результаты работы программы зависят также от версии Windows. Поскольку многопоточный код в различных средах может вести себя по-разному, никогда не следует полагаться на характеристики выполнения программы, достигнутые в одной среде. Например, в предыдущем примере было бы ошибкой предполагать, что низкоприоритетный поток будет всегда выполняться в течение хотя бы небольшого отрезка времени перед тем, как завершится поток с высоким приоритетом.

Например, в другой среде высокоприоритетный поток может завершиться еще до того, как поток с низким приоритетом хотя бы раз использует свой квант времени.

**Синхронизация**

При использовании в программе нескольких потоков иногда необходимо координировать их выполнение. Процесс координации потоков называется синхронизацией [4]. К синхронизации прибегают в тех случаях, когда двум или большему числу потоков необходимо получить доступ к общему ресурсу, который в каждый момент времени может использовать только один поток. Например, когда один поток выводит данные в файл, в это самое время второй поток должен быть лишен возможности выполнять аналогичные действия. Синхронизация необходима и в других ситуациях. Например, один поток ожидает, пока не произойдет определенное событие, а произойдет это событие или нет зависит от другого потока. В этом случае необходимо иметь средство, которое бы удерживало пер-

вый поток в состоянии приостановки до тех пор, пока не произойдет ожидаемое им событие. После этого "спящий" поток должен возобновить выполнение.

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

быть синхронизирован. Синхронизация поддерживается ключевым словом lock. Поскольку синхронизация заложена в С# с самого начала разработки языка, применять ее довольно легко.

Формат использования инструкции lock таков:

lock (object) {

// Инструкции, подлежащие синхронизации.

}

Здесь параметр object представляет собой ссылку на синхронизируемый объект. Если нужно синхронизировать только один элемент, фигурные скобки можно опустить. Инструкция lock гарантирует, что указанный блок кода, защищенный блокировкой для данного объекта, может быть использован только потоком, который получает эту блокировку. Все другие потоки остаются заблокированными до тех пор, пока блокировка не будет снята. А снята она будет лишь при выходе из этого блока.

В следующей программе демонстрируется синхронизация посредством управления

доступом к методу sumIt(), который суммирует элементы массива целочисленного типа (Пример 6):

//Использование инструкции lock для синхронизации доступа к объекту

using System;

using System.Threading; //необходимо подключать это пространство

//для многопоточных приложений

class SumArray

{

int sum;

public int sumIt(int[] nums)

{

lock (this)

{ //блокировка всего метода

sum = 0; //начальное значение суммы

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

{

sum += nums[i];

Console.WriteLine("Промежуточная сумма для потока " +

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

Thread.Sleep(10); //Разрешено переключение задач

}

return sum;

}

}

}

class MyThread

{

public Thread thrd;

int[] a;

int answer;

/\* Создаем один экземпляр класса SumArray для всех

экземпляров класса MyThread \*/

static SumArray sa = new SumArray();

//Создаем новый поток с помощью конструктора класса MyThread

public MyThread(string name, int[] nums)

{

a = nums;

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

thrd.Name = name;

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

}

//метод который будет выполняться при запуске потока

void run()

{

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

answer = sa.sumIt(a);

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

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

}

}

class Sync

{

public static void Main()

{

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

MyThread mt1 = new MyThread("Потомок №1", a);

MyThread mt2 = new MyThread("Потомок №2", a);

mt1.thrd.Join();

mt2.thrd.Join();

Console.ReadKey();

}

}

Результаты работы программы приведены на рисунке 23.

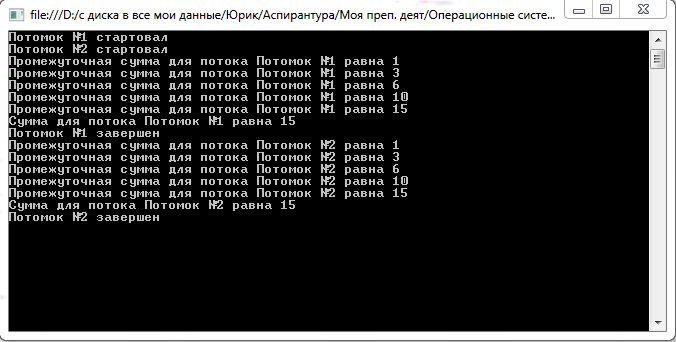


Рисунок 23 – Окно с результатами программы

Как видно по результатам, оба потока правильно подсчитали сумму чисел. В данной программе создается три класса. Первому присвоено имя SumArray. В нем определен метод sumIt(), который суммирует элементы целочисленного массива. Во втором классе, MyThread, используется static - объект sa типа SumArray. Таким образом, все объекты типа MyThread совместно используют только один объект типа SumArray. Этот объект служит для получения суммы целочисленных элементов массива. Обратите внимание на то, что промежуточная сумма хранится в поле sum класса SumArray. Следовательно, если два потока будут использовать метод sumIt() одновременно, каждый из них попытается хранить в поле sum "свою" промежуточную сумму. Поскольку это приведет к ошибочным результатам, доступ к методу sumIt() необходимо синхронизировать. Оба потока создаются в классе Sync, который дает им "путевку в жизнь", т.е. направляет на вычисление суммы элементов целочисленного массива. В методе sumIt() инструкция lock предотвращает одновременное его использование различными потоками. Обратите внимание на то, что в этой инструкции блокировки в качестве синхронизируемого объекта служит ссылка this. Обычно именно так выполнятся инструкция lock, если блокируется вызывающий объект. Метод Sleep() вызывается специально для того, чтобы разрешить переключение задач (если оно возможно), но в данном случае это невозможно. Поскольку код метода sumIt() заблокирован, он доступен одновременно только для одного потока. Следовательно, после того, как начнет выполняться второй поток, он не войдет в метод sumIt() до тех пор, пока его (метод) полностью не отработает первый дочерний поток. Тем самым гарантируется получение корректного результата. Чтобы до конца понять действие инструкции lock, попробуйте удалить ее из тела метода sumIt(). В этом случае метод sumIt() окажется без синхронизации, и любые потоки смогут использовать его одновременно для одного и того же объекта. Дело в том, что в поле sum хранится промежуточная сумма, и содержимое этого поля будет изменяться каждым потоком, который вызывает метод sumIt(). Таким образом, если два потока одновременно вызовут метод sumIt() для одного и того же объекта, будут

получены некорректные результаты, поскольку без синхронизации переменная sum будет отражать смешанный результат суммирования обоих потоков.

Действие инструкции lock.

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

2) Другие потоки, пытающиеся запросить блокировку для того же объекта, перейдут в состояние ожидания и будут находиться в нем до тех пор, пока код не будет разблокирован.

3) Объект разблокируется, когда поток выходит из заблокированного блока. Необходимо также отметить, что инструкция lock должна использоваться только для объектов, которые определены либо как private -, либо как internal-объекты. В противном случае внешние по отношению к вашей программе потоки смогут получать блокировку и не снимать ее.

Несмотря на то что блокирование кода метода, как показано в предыдущей программе, не представляет большой сложности и является эффективным средством синхронизации, оно, к сожалению, работает не во всех случаях. Например, нужно получить синхронизированный доступ к методу класса, который создавали не вы лично, а в нем самом средств синхронизации изначально не предусмотрено. Такая ситуация возможна в том случае, если вы хотите использовать класс, который написан сторонней организацией, и невозможно получить доступ к его исходному коду. В этом случае, очевидно, вам не удастся добавить инструкцию lock в соответствующий метод класса. Как тогда синхронизировать доступ к объекту такого класса? К счастью, есть очень простое решение описанной проблемы: заблокировать доступ к объекту из внешнего (по отношению к объекту) кода, указав этот объект в инструкции lock. Например, рассмотрим альтернативную реализацию предыдущей программы. Обратите внимание на то, что код в методе sumIt() больше не блокируется. Теперь блокируются обращения к методу sumIt() внутри класса MyThread (Пример 7).

//Еще один способ Использования инструкции lock для синхронизации доступа к объекту

using System;

using System.Threading; //необходимо подключать это пространство

//для многопоточных приложений

class SumArray

{

int sum;

public int sumIt(int[] nums)

{

sum = 0; //начальное значение суммы

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

{

sum += nums[i];

Console.WriteLine("Промежуточная сумма для потока " +

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

Thread.Sleep(10); //Разрешено переключение задач

}

return sum;

}

}

class MyThread

{

public Thread thrd;

int[] a;

int answer;

/\* Создаем один экземпляр класса SumArray для всех

экземпляров класса MyThread \*/

static SumArray sa = new SumArray();

//Создаем новый поток с помощью конструктора класса MyThread

public MyThread(string name, int[] nums)

{

a = nums;

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

thrd.Name = name;

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

}

//метод который будет выполняться при запуске потока

void run()

{

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

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

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

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

}

}

class Sync

{

public static void Main()

{

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

MyThread mt1 = new MyThread("Потомок №1", a);

MyThread mt2 = new MyThread("Потомок №2", a);

mt1.thrd.Join();

mt2.thrd.Join();

Console.ReadKey();

}

}

Здесь блокируется обращение к методу sa.sumIt(), а не код в самом методе sumIt(). Это реализуется таким способом:

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

При выполнении этой программы будут получены такие же корректные результаты, как и в предыдущем примере.

**Блокирование статического метода**

Поскольку блокировка работает по отношению к объекту, то на первый взгляд может показаться, что невозможно заблокировать код static-метода, поскольку не существует объекта, для которого требуется выполнить блокировку. В действительности все обстоит иначе. Чтобы заблокировать static-метод, достаточно использовать инструкцию lock в следующем формате:

lock (typeof(class)){

//Блокируемый код

}

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

**Класс Monitor и инструкция lock**

Ключевое слово lock - это не что иное, как сокращенный вариант использования средств синхронизации, определенных в классе Monitor, принадлежащем пространству имен System.Threading. В классе Monitor определено несколько методов синхронизации. Например, чтобы получить возможность блокировки для некоторого объекта, вызовите метод Enter(), а чтобы снять блокировку - метод Exit (). Эти методы имеют следующий формат:

public static void Enter(object *syncOb*)

public static void Exit(object *syncOb*)

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

public static bool TryEnter (object *syncOb*)

Метод возвращает значение true, если вызывающий поток получает блокировку для объекта *syncOb*, и значение false в противном случае. Если заданный объект недоступен, вызывающий поток будет ожидать до тех пор, пока он не станет доступным. В классе Monitor определены еще три метода: Wait(), Pulse() и PulseAll().

**Взаимодействие потоков с помощью методов Wait(), Pulse() и PulseAll()**

Рассмотрим следующую ситуацию. Поток (назовем его T) выполняет содержимое lock-блока и требует доступ к ресурсу (назовем его R), который временно недоступен. Что делать потоку 2? Если поток T войдет в цикл опроса в ожидании доступности ресурса R, он свяжет объект, блокируя доступ к нему другим потокам. Это решение трудно назвать оптимальным, поскольку оно аннулирует преимущества программирования в многопоточной среде. Будет лучше, если поток T временно откажется от "претензий" на объект, позволив другому потоку выполнить свою работу. Когда же ресурс R станет доступным, поток T можно уведомить об этом, и он возобновит выполнение. Такой подход опирается на межпоточные средства общения, которые позволяют одному потоку уведомить другой о том, что он блокируется, а затем первого поставить в известность о том, что он может возобновить выполнение. С# поддерживает межпоточное взаимодействие с помощью методов Wait(), Pulse() и PulseAll(). Методы Wait(), Pulse() и PulseAll() определены в классе Monitor. Эти методы можно вызывать только внутри lock-блока кода. Когда выполнение потока временно блокируется, вызывается метод Wait() т.е. он переходит в режим ожидания ("засыпает") и снимает блокировку с объекта, позволяя другому потоку использовать

этот объект. Позже, когда другой поток входит в аналогичное состояние блокирования и вызывает метод Pulse() или PulseAll(), "спящий" поток "просыпается". Обращение к методу Pulse() возобновляет выполнение потока, стоящего первым в очереди потоков, пребывающих в режиме ожидания. Обращение к методу PulseAll() сообщает о снятии блокировки всем ожидающим потокам.

Вот два наиболее корректных формата метода Wait():

public static bool Wait(object *waitOb*)

public static bool Wait(object *waitOb*, int milliseconds)

Первый формат означает ожидание до уведомления. Второй - подразумевает ожидание до уведомления или до истечения периода времени, заданного в миллисекундах. В обоих случаях параметр *waitOb* задает объект, к которому ожидается доступ.

Форматы использования методов Pulse() и PulseAll() таковы:

public static void Pulse(object *waitOb*)

public static void PulseAll(object *waitOb*)

Здесь параметр *waitOb* означает объект, освобождаемый от блокировки. Если метод Wait(), Pulse() или PulseAll() вызывается из кода, который находится вне lock-блока, генерируется исключение типа SynchronizationLockException.

**Взаимоблокировка**

При разработке многопоточных программ необходимо позаботиться о том, чтобы во время их выполнения не создалась тупиковая ситуация, вызванная взаимоблокировкой. При взаимоблокировке (deadlock) один поток ожидает, пока другой не выполнит некоторое действие, но в то же время второй поток ожидает действия первого. Таким образом, оба потока приостановлены, ожидая друг друга, и ни один из них не выполняется. Казалось бы, избежать взаимоблокировки нетрудно, но это не совсем так. Например, взаимоблокировка может возникнуть в ответвлениях программы. Зачастую причину взаимоблокировки нелегко понять, просто изучая исходный код программы, поскольку одновременно выполняющиеся потоки могут сложным образом взаимодействовать во время работы. Чтобы избежать взаимоблокировки, необходимо очень аккуратно подходить к программированию и тщательно тестировать написанный код. Как правило, если многопоточная программа вдруг "виснет", то наиболее вероятная причина этого взаимоблокировка.

Используя атрибут MethodImplAttribute, можно синхронизировать метод целиком.

Этот подход можно рассматривать как альтернативу инструкции lock в случаях, когда необходимо заблокировать все содержимое метода. Атрибут MethodImplAttribute определен в пространстве имен System.Runtime.CompilerServices. Конструктор, используемый для синхронизации, имеет такой вид:

public MethodImplAttribute(MethodImplOptions *opt)*

Здесь параметр *opt* задает атрибут реализации. Для синхронизации метода задайте атрибут MethodImplOptions.Synchronized. Этот атрибут обеспечивает блокировку всего метода.

**Приостановка, возобновление и завершение выполнения потоков**

Иногда выполнение потока необходимо приостановить. Например, поток можно использовать для отображения времени суток. Если пользователь желает убрать часы с экрана, соответствующий поток можно приостановить. Позже, когда необходимость в часах появится снова, выполнение приостановленного потока можно возобновить. В любом случае приостановить и возобновить поток иногда необходимо. Иногда нужно и вовсе завершить выполнение потока. Завершение выполнения потока отличается от приостановки тем, что завершенный поток удаляется из системы, и его выполнение не может быть возобновлено в дальнейшем.

Для приостановки потока используйте метод Thread.Suspend(), а для его возобновления - метод Thread.Resume(). Форматы использования этих методов таковы:

public void Suspend()

public void Resume()

Если вызывающий поток находится не в подходящем для вызываемого метода состоянии, генерируется исключение типа ThreadStateException. Такие последствия может иметь, например, попытка возобновить поток, который не был приостановлен. Чтобы завершить поток, используйте метод Thread.Abort(). Самый простой формат его использования выглядит так:

public void Abort()

Метод Abort() генерирует исключение типа ThreadAbortException для потока, из которого этот метод вызван. Это исключение и заставляет поток завершиться. Кроме того, то же самое исключение может быть перехвачено программным кодом (с автоматической его регенерацией для завершения потока). Однако следует учитывать, что метод Abort() не всегда способен немедленно остановить выполнение потока, поэтому, если важно, чтобы поток был завершен до продолжения вашей программы, необходимо сопроводить вызов метода Abort() вызовом метода Join(). В некоторых (довольно редких) случаях метод Abort() не в состоянии завершить поток. Это возможно в ситуации, когда finally-блок включен в бесконечный цикл. В следующем примере демонстрируется приостановка, возобновление и завершение выполнения потока (Пример 8):

// Приостановка, возобновление и завершение потока.

using System;

using System.Threading;

class MyThread

{

public Thread thrd;

public MyThread(string name)

{

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

thrd.Name = name;

thrd.Start();

}

// Это входная точка для потока

void run()

{

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

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

{

Console.Write(i + " ");

if ((i % 10) == 0)

{

Console.WriteLine();

Thread.Sleep(250);

}

}

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

}

}

class SuspendResumeStop

{

public static void Main() {

MyThread mt1 = new MyThread ("Мой поток");

Thread.Sleep (1000); // Разрешаем стартовать

// дочернему потоку

mt1.thrd.Suspend ();

Console.WriteLine ("Приостановка выполнения потока.");

Thread.Sleep (1000);

mt1.thrd.Resume ();

Console.WriteLine ("Возобновление выполнения потока.");

Thread.Sleep (1000);

mt1.thrd.Suspend ();

Console.WriteLine ("Приостановка выполнения потока.");

Thread.Sleep (1000);

mt1.thrd.Resume ();

Console.WriteLine ("Возобновление выполнения потока.");

Thread.Sleep (1000);

Console.WriteLine ("завершение выполнения потокаю.");

mt1.thrd.Abort ();

mt1.thrd.Join (); // Ожидаем завершения выполнения потока

Console.WriteLine ("Основной поток завершен");

Console.ReadKey();

}

}

Результат работы программы показан на рисунке 24.

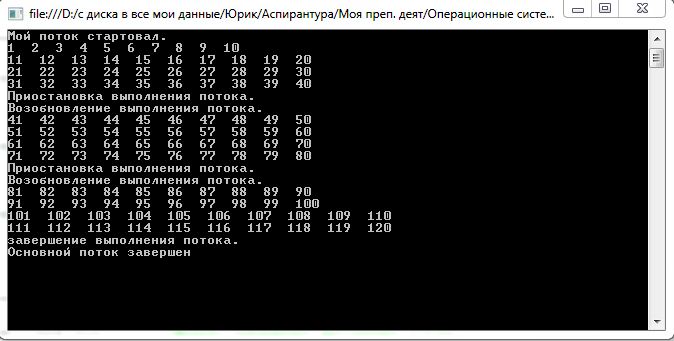


Рисунок 24 – Окно с результатами программы

В некоторых случаях удобней использовать второй формат метода Abort( ):

public void Abort(object *info*)

Здесь параметр *info* содержит информацию, которую необходимо передать в поток при его завершении. Эта информация доступна через свойство ExceptionState класса ThreadAbortException. Этот формат можно использовать для передачи потоку кода завершения, что и демонстрируется в следующем примере (Пример 9):

// Использование метода Abort(object)

using System;

using System.Threading;

class MyThread

{

public Thread thrd;

public MyThread(string name)

{

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

thrd.Name = name;

thrd.Start();

}

// Это входная точка для потока

void run(){

try

{

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

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

{

Console.Write(i + " ");

if ((i % 10) == 0)

{

Console.WriteLine();

Thread.Sleep(250);

}

}

Console.WriteLine(thrd.Name + " завершился нормально");

}

catch (ThreadAbortException exc)

{

Console.WriteLine("Выполнение потока прервано, код завершения= " + exc.ExceptionState);

}

}

}

class UseAltAbort{

public static void Main() {

MyThread mt1 = new MyThread ("Мой поток");

Thread.Sleep (1000); // Разрешаем стартовать

// дочернему потоку

Console.WriteLine("Прерывание выполнения потока");

mt1.thrd.Abort(100);

mt1.thrd.Join (); // Ожидаем завершения выполнения потока

Console.WriteLine ("Основной поток завершен");

Console.ReadKey();

}

}

Результат работы программы показан на рисунке 25.

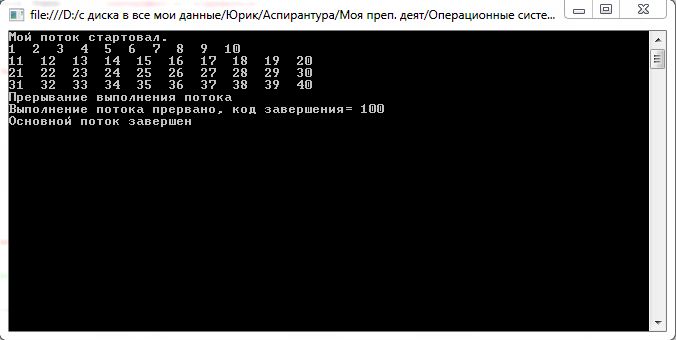


Рисунок 25 – Окно с результатами программы

Как подтверждают результаты выполнения этой программы, методу Abort() передано число 100. Это значение затем можно прочитать в свойстве ExceptionState класса исключения ThreadAbortException, перехватываемого потоком при завершении.

**Отмена действия метод Abort()**

Поток может переопределить запрос на прерывание выполнения. Для этого поток должен перехватить исключение типа ThreadAbortException, а затем вызвать метод ResetAbort(). Это защищает исключение от автоматической регенерирации по окончании его обработки потоком. Метод ResetAbort() объявляется следующим образом:

public static void ResetAbort()

Обращение к методу ResetAbort() может оказаться неудачным, если поток не имеет соответствующего уровня доступа, чтобы отменить прерывание выполнения. Использование метода ResetAbort() демонстрируется в следующей программе (Пример 10):

// Использование метода Abort(object)

using System;

using System.Threading;

class MyThread

{

public Thread thrd;

public MyThread(string name)

{

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

thrd.Name = name;

thrd.Start();

}

// Это входная точка для потока

void run()

{

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

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(

"Прерывание отменено! Код завершения = " + exc.ExceptionState);

Thread.ResetAbort();

}

else

Console.WriteLine(

"Выполнение потока прервано, код завершения = "

+ exc.ExceptionState);

}

}

Console.WriteLine(thrd.Name + " завершился нормально");

}

}

class ResetAbort{

public static void Main() {

MyThread mt1 = new MyThread ("Мой поток");

Thread.Sleep (1000); // Разрешаем стартовать

// дочернему потоку

Console.WriteLine("Прерывание выполнения потока");

mt1.thrd.Abort(0); //Это не остановит выполнение потока

Thread.Sleep(1000);

Console.WriteLine("Прерывание выполнения потока");

mt1.thrd.Abort(100);

mt1.thrd.Join (); // Ожидаем завершения выполнения потока

Console.WriteLine ("Основной поток завершен");

Console.ReadKey();

}

}

Результат работы программы показан на рисунке 26.

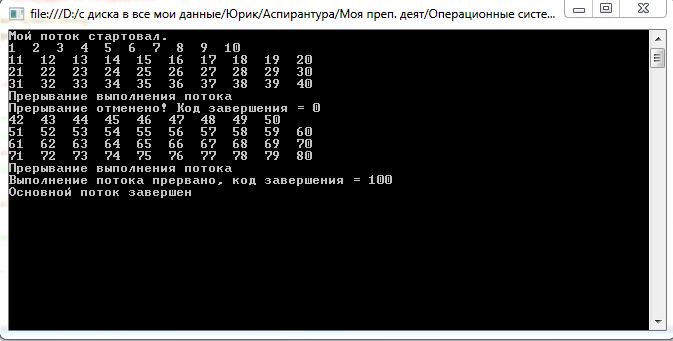


Рисунок 26 – Окно с результатами программы

Мы убедились, что, если метод Abort() вызывается с аргументом, значение которого равно нулю, то посредством вызова метода ResetAbort() запрос на прерывание отменяется, и поток продолжает выполняться. Любое другое значение этого аргумента останавливает выполнение потока.

**Определение состояния потока**

Состояние потока можно получить из свойства ThreadState, определенного в классе Thread:

public Thread ThreadState{ get; }

Состояние потока возвращается как значение, определенное перечислением ThreadState. В нем определены такие значения:

ThreadState.Aborted ThreadState.AbortRequested

ThreadState.Background ThreadState.Running

ThreadState.Stopped ThreadState.StopRequested

ThreadState.Suspended ThreadState.SuspendRequested

ThreadState.Unstarted ThreadState.WaitSleepJoin

Одно из перечисленных выше значений требует пояснения. В состояние, представленное значением ThreadState.WaitSleepJoin, поток входит, ожидая результатов вызова метода Wait(), Sleep() или Join().

**Использование основного потока**

Все С# программы имеют по крайней мере один поток управления, именуемый основным потоком, который автоматически создается в начале выполнения программы. Основной поток обрабатывается подобно всем остальным потокам. Чтобы получить доступ к основному потоку, необходимо иметь объект класса Thread, который его представляет. Это реализуется с помощью свойства CurrentThread, которое является членом класса Thread. Формат его использования таков:

public static Thread CurrentThread{ get; }

Это свойство возвращает ссылку на поток, в котором оно опрашивается. Следовательно, если при выполнении основного потока обратиться к свойству CurrentThread, мы получим ссылку на основной поток. Имея такую ссылку, можно управлять основным потоком, как и любым другим. В следующей программе демонстрируется, как получить ссылку на основной поток, узнать его имя и приоритет, а также задать новое имя и приоритет (Пример 11):

// Управление основным потоком

using System;

using System.Threading;

class UseMain

{

public static void Main()

{

Thread thrd;

//Получаем ссылку на объект основного потока

thrd = Thread.CurrentThread;

//Отображаем имя основного потока

if (thrd.Name == null)

Console.WriteLine("Основной поток не имеет имени");

else

Console.WriteLine("Имя основного потока: " + thrd.Name);

//Отображаем приоритет основного потока

Console.WriteLine("Приоритет: " + thrd.Priority);

Console.WriteLine();

//Задаем имя и приоритет

Console.WriteLine("Установка имени и приоритета\n");

thrd.Name = "Основной поток";

thrd.Priority = ThreadPriority.AboveNormal;

Console.WriteLine("У основного потока теперь есть имя: " + thrd.Name);

Console.WriteLine("Идентификатор основного потока: " + thrd.ManagedThreadId);

Console.WriteLine("Контекст основного потока: " + thrd.ExecutionContext);

Console.WriteLine("Приоритет теперь таков: " + thrd.Priority);

Console.ReadKey();

}

}

Результат работы программы показан на рисунке 27.

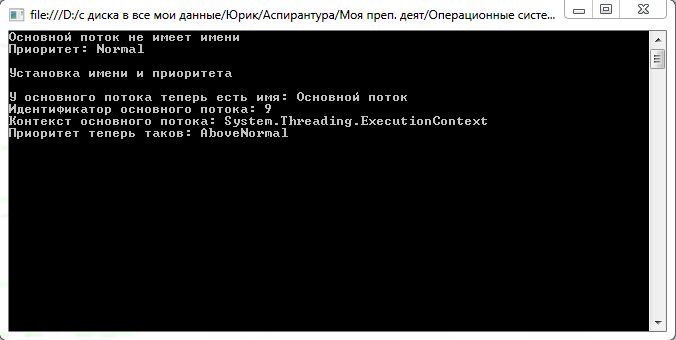


Рисунок 27 – Окно с результатами программы

Необходимо соблюдать осторожность при выполнении операций над основным потоком. Например, если обращение к методу Join() thrd.Join(); добавить в конец метода Main(), программа никогда не завершится, поскольку она будет ожидать, пока не завершится основной поток!

Ключ к эффективному использованию многопоточности лежит в "параллельном" мышлении (в противоположность последовательному). Например, если в вашей программе предполагается функционирование двух параллельно работающих подсистем, организуйте их в виде отдельных потоков. Но если создать слишком много потоков, реальное быстродействие программы может пострадать. Помните: каждое переключение контекста требует определенных расходов системных ресурсов. При большом количестве потоков на изменение контекста будет потрачено больше процессорного времени, чем на выполнение самой программы!

**Запуск отдельной задачи**

Несмотря на то что чаще всего в С#-программировании используется поточно-ориентированная многозадачность, в соответствующих случаях возможно применение и процессно-ориентированной многозадачности. В многозадачной среде, ориентированной на процессы, вместо запуска в той же программе еще одного потока, одна программа запускает на выполнение другую программу. В С# это реализуется с использованием класса Process, который определен в пространстве имен System.Diagnostics. Самый простой способ запустить другую программу - использовать метод Start(), определенный в классе Process. Вот один из простейших его форматов:

public static Process Start(string name)

Здесь параметр name означает имя выполняемого файла или файла, связанного с

выполняемым. При завершении созданного процесса вызовите метод Close(), чтобы освободить память, занимаемую этим процессом. Метод Close() объявляется так:

public void Close()

Завершить процесс можно двумя способами. Если процесс представляет собой GUI-приложение, ориентированное на выполнение под управлением Windows, то для завершения такого процесса достаточно вызвать метод CloseMainWindow(), определяемый следующим образом:

public bool CloseMainWindow()

Этот метод посылает процессу сообщение, предписывающее ему остановиться. Метод возвращает значение true, если посланное сообщение получено. Метод возвращает значение false, если данное приложение не является GUI-программой или не имеет главного окна. Более того, метод CloseMainWindow() - лишь запрос на прекращение работы. Если он игнорируется приложением, завершение не состоится. Для безусловного завершения процесса необходимо вызвать метод Kill():

public void Kill()

Однако использовать метод Kill() нужно с большой осторожностью. Он обеспечивает бесконтрольное завершение процесса. Любые несохраненные данные, связанные с этим процессом, будут, скорее всего, утеряны. Ожидать завершения процесса можно с помощью метода WaitForExit(). Его два возможных формата таковы:

public void WaitForExit()

public bool WaitForExit(int milliseconds)

Первый формат позволяет ожидать до тех пор, пока процесс не завершится, а второй - лишь в течение заданного числа миллисекунд. Метод WaitForExit(), используемый во втором формате, возвращает значение true, если в течение заданного промежутка времени процесс завершился, и значение false, если он все еще выполняется. Следующая программа демонстрирует создание процесса и ожидание его завершения. Программа запускает стандартную утилиту Windows calc.exe (Пример 12).

// Запуск нового процесса

using System;

using System.Diagnostics;

class StartProcess

{

public static void Main()

{

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

Console.WriteLine("Новый процесс стартовал");

newProc.WaitForExit();

newProc.Close(); //Освобождаем ресурсы системы

Console.WriteLine("Новый процесс завершился");

}

}

При выполнении этой программы запускается текстовый калькулятор calc.exe, и на

экране появится сообщение "Новый процесс стартовал". Затем программа будет ожидать, пока вы не закроете утилиту calc.exe. После завершения работы calc.exe отобразится последнее сообщение "Новый процесс завершился".

## **14.3 Перечень индивидуальных заданий**

Изучите теоретический материал и выполните все примеры программ, приведенные в лабораторной работе. Выполните индивидуальное задание согласно варианту (таблица 1).

Таблица 1 – Варианты заданий

|  |  |
| --- | --- |
| Вариант | № задания |
| 1 | 1, 2 |
| 2 | 3, 4 |
| 3 | 5, 6 |
| 4 | 7, 8 |
| 5 | 1, 3 |
| 6 | 2, 4 |
| 7 | 3, 6 |
| 8 | 3, 5 |
| 9 | 4, 8 |
| 10 | 1, 8 |

1. Составить программу на языке С# в которой работают параллельно два потока. В первом потоке происходит суммирование чисел от 1 до 1000, а во втором потоке происходит чтение 1000 записей из текстового файла. Запуск потоков осуществить при помощи конструктора класса.

2. Переделать пример 12, так чтобы после запуска калькулятора проходило 10 секунд и процесс был завершен с помощью метода Kill() с выдачей на экран соответствующего уведомления пользователю. Пока будет запущен процесс calc.exe необходимо приостановить выполнение основного потока вашего приложения.

3. Составить программу на языке С# в которой работают параллельно два потока. В первом потоке необходимо создать подключение к базе данных и считывать все записи из одной таблицы этой базы данных, а в другом потоке необходимо параллельно вычислять сумму чисел от 1 до 1000.

4. В примере 6 изменить приоритет первого потока установив значение ThreadPriority.Highest, а второго ThreadPriority.Lowest; Выполнить программу и объяснить результаты. В примере 12 запустить calc.exe как фоновый процесс.

5. В примере 10 необходимо предусмотреть ввод пользователем с клавиатуры кода завершения потока (число от 1 до 100) – это организовать в отдельном потоке, в то время как в другом потоке должны происходить вычисления.

6. В примере 10 необходимо предусмотреть ввод пользователем с клавиатуры кода завершения потока (число от 20 до 50) – это организовать в отдельном потоке, в то время как в другом потоке должны происходить вычисления в цикле суммы чисел от 1 до 1000.

7. Организовать в программе три потока. Первый поток должен создавать текстовый файл на жестком диске. А вторые два потока должны по очереди записывать в этот файл результаты вычислений суммы чисел от 1 до 999. Необходимо синхронизировать доступ к файлу второму и третьему потоку, чтобы файл был разделяемым ресурсом.

8. Организовать в программе три потока. Первый поток должен создавать текстовый файл на жестком диске. Второй поток должен записывать в этот файл промежуточные значения переменной (четные значения), в которой суммируются числа от 1 до 500. Третий поток должен считывать четные значения из файла и также их суммировать и итоговую сумму вывести на экран. Все потоки должны быть запущены на выполнение одновременно. Необходимо синхронизировать доступ к файлу второму и третьему потоку, чтобы файл был разделяемым ресурсом.

## **14.4 Оформление отчёта**

При оформлении отчёта студент для каждой решённой задачи должен привести:

* + условие задания;
  + алгоритм решения задачи;
  + исходный код программы;
  + результаты тестирования (где возможно в сравнении с ручным просчётом решения задачи);
  + вывод о корректности работы реализованных алгоритмов.

## **14.5 Контрольные вопросы**

1. В чем заключаются основы многопоточности?

2. В каком пространстве имен определены классы, поддерживающие многопоточное программирование?

3. Опишите методы и свойства класса Thread?

4. Для чего предназначен метод Join()?

5. Как программно изменить приоритет потока?

6. В каких случаях прибегают к синхронизации потоков?

7. Для чего используется инструкция lock?

8. Как производится блокирование статического метода?

9. Для чего предназначен класс Monitor?

10. Приведите примеры использования методов Wait(), Pulse() и PulseAll().