**C# как язык параллельного программирования.**

**Библиотека System.Threading.**

Курсовая работа

по дисциплине Алгоритмы и языки

параллельного программирования

студента физико-математического

факультета 4 курса группы “ВГ”

Котова Виталия Евгеньевича

Научный руководитель

Старший преподаватель кафедры информатика

Каменская Наталья Евгеньевна

# **ОГЛАВЛЕНИЕ**

[ОГЛАВЛЕНИЕ 2](#_Toc389734398)

[ВВЕДЕНИЕ 3](#_Toc389734399)

[ГЛАВА 1. ОСНОВЫ МНОГОПОТОЧНОЙ ОБРАБОТКИ 4](#_Toc389734400)

[1.1. Создание и запуск потока 4](#_Toc389734401)

[1.2. Определение момента окончания потока 5](#_Toc389734402)

[1.3. Передача аргумента потоку 6](#_Toc389734403)

[1.4. Приоритеты потоков 7](#_Toc389734404)

[1.5. Свойство IsBackground 8](#_Toc389734405)

[1.6. Прерывание потока 8](#_Toc389734406)

[ГЛАВА 2. СИНХРОНИЗАЦИИЯ ПОТОКОВ 10](#_Toc389734407)

[2.1. Понятие синхронизации 10](#_Toc389734408)

[2.2. Взаимоблокировка и состояние гонки 11](#_Toc389734409)

[ГЛАВА 3. Полная реализация решения задачи 13](#_Toc389734410)

[3.1. Создание класса Forest 13](#_Toc389734411)

[3.2. Класс LifeSaver 14](#_Toc389734412)

[3.3. Классы Father, Mother, GrandFather, GrandMother 14](#_Toc389734413)

[3.4. Класс Girl 15](#_Toc389734414)

[ЗАКЛЮЧЕНИЕ 16](#_Toc389734415)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 17](#_Toc389734416)

# ВВЕДЕНИЕ

Среди многих свойств языка С# особое место принадлежит поддержке многопоточного программирования. Многопоточное программирование опирается на целый ряд средств, предусмотренных для этой цели в самом языке С#, а также на классы, определенные в среде .NET Framework. Благодаря встроенной в С# поддержке многопоточной обработки сводятся к минимуму или вообще устраняются многие трудности, связанные с организацией многопоточной обработки в других языках программирования.

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

using System.Threading;

Рассмотрим средства многопоточного программирования языка на примере решения следующей задачи.

Однажды, в темном-темном лесу потерялась девочка. Размер леса – N x N. Ходит-ходит девочка по лесу со скоростью V в произвольном направлении, но найти край леса не может. Девочка просто хаотично двигается по лесу, и при этом кричит «Ауу» в надежде, что кто-нибудь ее услышит. Известно, что по лесу также двигаются еще несколько объектов со скоростью 2V – мама, папа, дедушка и бабушка девочки. Если кто-нибудь из них услышит девочку, то девочка будет спасена. Услышать ее могут, если между девочкой и слушателем не более четырех квадратов леса. Если она самостоятельно достигнет края леса, то также будет спасена. Необходимо создать программную систему, в которой реализуется данная ситуация, при этом желательно визуализировать процесс поиска, достаточно использовать псевдографику.

Цель данной курсовой работы рассмотреть средства библиотеки System.Threading на примере решения данной задачи.

1. ОСНОВЫ МНОГОПОТОЧНОЙ ОБРАБОТКИ

## Создание и запуск потока

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.

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. Приведем полный пример создания и запуска потока.

public Thread movement;

movement = new Thread(MovementGirls);

movement.Start();

где MovementGirls метод реализующий алгоритм передвижения девочки по лесу.

## Определение момента окончания потока

Нередко оказывается полезно знать, когда именно завершается поток. 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.

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. Следующий фрагмент кода показывает использование метода Join().

Girl girl = new Girl();

girl.movement.Join();

где объект girl класса Girl имеет поток movement, у которого и вызывается метод 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. 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.

## Приоритеты потоков

У каждого потока имеется свой приоритет, который отчасти определяет, насколько часто поток получает доступ к ЦП. 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.

## Свойство IsBackground

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.

public bool IsBackground { get; set; }

Для того чтобы сделать поток фоновым, достаточно присвоить логическое значение true свойству IsBackground. А логическое значение false указывает на то, что поток является приоритетным. Пример использования свойства IsBackground:

private Thread searching;

searching = new Thread(Movement);

searching.IsBackground = true;

## 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. 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.

1. СИНХРОНИЗАЦИИЯ ПОТОКОВ

## Понятие синхронизации

Когда используется несколько потоков, то иногда приходится координировать их действия. 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. 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. Пример использования оператора блокировки приведен ниже:

private void MovementGirls()

{

Move move;

while (!Found)

{

lock (Forest.forest)

{

//необходимые операции с объектом Forest.forest

}

}

}

## 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.

На первый взгляд избежать взаимоблокировки нетрудно, но на самом деле не все так просто, ведь взаимоблокировка может возникать окольными путями. 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.

1. Реализация решения задачи

## Создание класса Forest

Создадим класс Forest в котором будет реализован лес.

struct ForestPoint

{

public bool heardAyy; //истина если в точке леса

//слышна «Ауу»!

public int personID;

}

static class Forest

{

private static int \_size = 50;//размер леса

public static ForestPoint[,] forest =

new ForestPoint[\_size, \_size];

public static int Size

{

get { return \_size; }

}

public static void Print()

{

for (int i = 0; i < \_size; i++)

{

for (int j = 0; j < \_size; j++)

{

//выводим в консоль букву в

//соответствии c personID

}

}

}

public static void PrintChanges()

{

Thread.Sleep(30);

Console.Clear();

Forest.Print();

}

}

где метод Print() выводит в консоль состояние леса.

## Класс LifeSaver

В класс LifeSaver реализованы методы которые отвечают за поиск девочки, а также следующие поля: \_x, \_y – первоначальные координаты спасателя; \_speed – скорость передвижения спасателя по лесу; \_girl – ссылка на объект класса Girl; \_idLifeSaver – номер спасателя; searching – ссылка на объект класса Thread. Конструктор этого класса приведен ниже.

public LifeSaver(int id, int yStart, int xStart, Girl girl,

int speed)

Опишем методы класса LifeSaver.

private void Movement()

в этом методе осуществляется движение спасателя по лесу.

protected virtual void MovementAlgorithm(int speed

ref int y, ref int x)

реализует алгоритм движения спасателя. Алгоритм представляет собой хаотичное передвижение в пределах размера леса.

## Классы Father, Mother, GrandFather, GrandMother

Классы Father, Mother, GrandFather, GrandMother наследуют класс LifeSaver. В классах Mother и GrandMother перегружается метод MovementAlgorithm(), реализующий алгоритм движения спасателя.

## Класс Girl

В классе Girl реализован объект девочка. Класс имеет следующие поля: \_x, \_y – первоначальные координаты девочки; \_region – размер области где слышно девочку; movement – ссылка на объект класса Thread.

Класс Girl имеет следующий конструктор:

public Girl()

{

\_random = new Random();

\_region = 4;

\_x = \_random.Next(1, Forest.Size - 2);

\_y = \_random.Next(1, Forest.Size - 2);

lock (Forest.forest)

{

SetRegion(true);

Forest.PrintChanges();

}

movement = new Thread(MovementGirls);

movement.Start();

}

Реализации этого класса имеет следующие методы:

private void MovementGirls()

этот метод осуществляется движение девочки по лесу.

private void MoveGirl\_and\_Region(Move moveTo)

перемещает девочку и ее голос по лесу.

# ЗАКЛЮЧЕНИЕ

В данной курсовой работе были рассмотрены средства библиотеки System.Threading на примере решения задачи, сформулированной во введении. Найти приложение реализующее решение задачи можно в [Disk]:\CourseWork\LostGirl\LostGirl\obj\x86\Debug\ LostGirl.exe.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Рихтер Дж., CLR via C#. Программирование на платформе Microsoft .NET Framework 4.0 на языке C#, 2012. – 928 с.
2. Троелсен, Э., Язык программирования C# 5.0 и платформа .NET 4.5 6-е издание, 2012. – 1311 с.
3. Шилдт, Г., C# 4.0 Полное руководство, 2011. – 1056 с.
4. Microsoft Developer Network [Электронный ресурс] Режим доступа: http://msdn.microsoft.com/ru-ru/library/system.threading.aspx

Дата доступа: 10.05.2014.