Лабораторная работа №4 «Основы многопоточности»

Цель работы: Изучить способы создания нескольких потоков и влияние приоритета на исполнение потока.

Теоретическая часть

Основы многопоточности

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

Следует также запомнить, что в действительности потоки выполняются всё-таки не совсем параллельно. Дело в том, что процессор физически не может обрабатывать параллельно несколько инструкций или процессов. Однако его вычислительной мощи хватает настолько, что он может выполнять все операции по небольшому фрагменту по очереди, отводя на каждый такой фрагмент по очень маленькому кусочку времени, настолько, что кажется, будто все процессы в компьютере выполняются параллельно.

Точно такая же ситуация происходит и с потоками. Если в программе имеется 3 потока, то сначала выполняется кусочек кода из одного потока, потом кусочек кода из другого, затем – из третьего, после чего процессор снова переходит к какому-либо из двух других потоков. Выбор, какой поток необходимо назначить для выполнения в данный момент остаётся за процессором. Происходит это в доли миллисекунд, поэтому происходит ощущение параллельной работы потоков.

Стандартно в проектах Visual Studio существует только один основной поток – в методе Main. Всё, что в нём выполняется – выполняется последовательно строка за строкой. Но при необходимости можно “распараллелить” выполняемые процессы при помощи потоков.

Язык C# имеет встроенную поддержку многопоточности, а среда .NET Framework предоставляет сразу несколько классов для работы с потоками, что в купе очень помогает гибко и правильно реализовывать и настраивать многопоточность в проектах.

В среде .NET Framework существует два типа потоков: основной и фоновый (вспомогательный). В целом отличие между ними одно – если первым завершится основной поток, то фоновые потоки в его процессе будут также принудительно остановлены, если же первым завершится фоновый поток, то это не повлияет на остановку основного потока – тот будет продолжать функционировать до тех пор, пока не выполнит всю работу и самостоятельно не остановится. Обычно при создании потока ему по-умолчанию присваивается основной тип.

Практическая часть

Для работы с потоками в C# необходимо подключить специальную директиву:

```
using System.Threading;
```

Именно она позволяет реализовывать необходимые потоки и их настройку. Любой поток в C# должен обязательно происходить в каком-либо методе или функции, поэтому для работы с новым потоком необходимо сначала создать для него, например, метод, который и будет точкой входа для этого потока. Для примера создадим пустой метод под именем *fornewthread*, который ничего не возвращает:

```
static void fornewthread()
{
// здесь должны находиться инструкции, которые будут выполняться в новом потоке
}
```

Теперь мы можем создать сам поток (например, в главном методе Main). Назовём его mythread:

```
Thread mythread = new Thread(fornewthread);
```

Для создания потока нужно вызвать делегат Thread, а также передать конструктору адрес метода (в скобках). Потоки в C# начинают выполняться не сразу после их инициализации. Каждый из созданных потоков необходимо сначала запустить. Делается это следующим образом: имя_потока.Start();. В случае из нашего примера строка запуска будет такой:

```
mythread.Start();
```

Теперь, при запуске программы, как только в Main выполнится метод Start(), начнёт работать вновь созданный поток. После того, как выполнился Start(), работа программы “распараллеливается”: один поток начинает выполнять код из метода fornewthread, а второй поток продолжает выполнять операции, которые остались ещё не выполненными в методе Main (после Start()). Естественно, это всё произойдёт, если в fornewthread или в Main имеется дополнительный код. Если же один из потоков выполнит работу раньше другого, то первый будет ожидать окончания выполнения работы последнего. А делать это он будет по причине, описанной выше: так как оба потока являются основными, то завершение какого-либо одного потока не влияет на завершение другого. Если бы один из наших потоков был фоновым и “задержался” бы с работой, то при окончании работы основного потока, принудительно завершился бы и он.

Иногда бывают такие сценарии, когда необходимо бывает приостановить поток, например для того, чтобы пропустить другие потоки и не мешать им выполнять свою работу, либо для снижения потребления процессорного времени.

Есть несколько способов остановки потоков:

```
Thread.Sleep(100);
```

В этой строке мы указали, что поток, к которому будет относиться данная инструкция, будет приостановлен на 100 миллисекунд. Как только заданное время пройдёт, поток снова возобновит свою работу и продолжит с места, на котором остановился. Число, заданное в скобках – это количество миллисекунд для задержки в идеале, на практике поток может возобновиться на несколько миллисекунд позже или раньше, это зависит не только от программы, но и от характеристик операционной системы.

В метод Sleep также можно передать и значение ноль:

```
Thread.Sleep(0);
```

В таком случае поток приостановится на положенный ему интервал времени (мы же помним, что все потоки выполняются по кусочку и по очереди, и при этом на каждый такой кусочек выделяется очень небольшое количество времени, но достаточное для того, чтобы выполнить некоторое количество действий), который он “отдаёт” любому другому готовому к

выполнению потоку с таким же приоритетом, как и у него (про приоритеты поговорим ниже). Если же не удастся найти ни один такой “ожидающий” поток, то выполнение текущего потока не будет приостановлено, и он продолжит свою работу.

В первой программе реализуем работу нескольких параллельных потоков. Для примера создадим проект C# с тремя потоками, каждый из которых будет выводить числа от 0 до 9.

```
static void mythread1()
{
        for (int i = 0; i < 10; i++)
                {
                Console.WriteLine("Поток 1 выводит " + i);
                }
}
static void mythread2()
{
        for (int i = 0; i < 10; i++)
        {
                Console.WriteLine("Поток 2 выводит " + i);
        }
}
static void Main(string[] args)
{
        Thread thread1 = new Thread(mythread1);
        Thread thread2 = new Thread(mythread2);
        thread1.Start(); thread2.Start();
        for (int i = 0; i < 10; i++)
        {
                Console.WriteLine("Поток 3 выводит " + i);
        }
        Console.ReadLine();
}
```

Работа с приоритетами 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.

```
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.

```
using System;
using System.Threading;

class MyThread
{
	public int Count;
	public Thread Thrd;
	static bool stop = false;
	static string currentName;
	public MyThread(string name)
	{
		Count = 0;
		Thrd = new Thread(this.Run);
		Thrd.Name = name;
		currentName = name;
	}
	// Начать выполнение нового потока,
	void Run()
	{
		Console.WriteLine("Поток " + Thrd.Name + "начат.");
		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 {
	static void Main()
	{
		MyThread mt1 = new MyThread("с высоким приоритетом");
		MyThread mt2 = new MyThread("с низким приоритетом");
		// Установить приоритеты для потоков.
		mt1.Thrd.Priority = ThreadPriority.AboveNormal;
		mt2.Thrd.Priority = ThreadPriority.BelowNormal;
		// Начать потоки,
		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);
	}
```

}

Конкретный результат процессорного времени 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.

Порядок выполнения работы

1. Ознакомиться с теоретической частью.
2. Повторить практические примеры и прокомментировать каждую строку кода так, чтобы было ясно что она делает.
3. Реализовать программу, использующую 3 потока: 1й поток должен выводить числа от 0 до 9, 2й поток – числа от 10 до 19, 3й поток числа от 100 до 120.
4. Установить потокам приоритеты Lowest, BelowNormal и Highest.
5. Сделать выводы о выделенном каждому потоку процессорном времени.