Лабораторная работа №9

Многопоточное программирование

Для разделения различных выполняемых приложений в операционных системах используются процессы. Потоки являются основными элементами, для которых операционная система выделяет время процессора; внутри процесса может выполняться более одного потока. Каждый поток поддерживает обработчик исключений, планируемый приоритет и набор структур, используемых системой для сохранения контекста потока во время его планирования. Контекст потока содержит все необходимые данные для возобновления выполнения (включая набор регистров процессора и стек) в адресном пространстве ведущего процесса потока.

Операционная система, поддерживающая приоритетную многозадачность, создает эффект одновременного выполнения нескольких потоков из нескольких процессов. Это выполняется путем последовательного распределения доступного времени процессора между требуемыми потоками. Текущий выполняемый поток приостанавливается по истечении выделенного времени; после этого запускается другой поток. При переключении от одного потока к другому контекст потока, работа которого была приостановлена, сохраняется, и загружается контекст следующего потока из очереди.

Количество выделяемого для потока времени определяется операционной системой и процессором. Поскольку это время мало, выполнение нескольких потоков будет происходить почти одновременно, даже при наличии одного процессора. В многопроцессорной системе выполняемые потоки распределяются между доступными процессорами.

Для работы с потоками в среде .NET имеется пространство имен System.Threading. Создать поток можно с помощью класса Thread, для этого следует передать делегат ThreadStart или ParameterizedThreadStart, связанный с программным кодом, который будет находиться под управлением потоком, в

конструктор класса Thread. С помощью делегата ParameterizedThreadStart можно передавать данные в потоковую процедуру. Поток будет запущен только после вызова метода Start(). Например:

```
Thread newThread = new Thread(new ThreadStart(threadWork.DoWork));
newThread.Start();
```

Процедура, на которую ссылается делегат, может быть статической, методом экземпляра класса:

```
public class Work
{
    public static void Main()
    {
        Thread newThread = new Thread(Work.DoWork);
        newThread.Start(42);
        Work w = new Work();
        newThread = new Thread(w.DoMoreWork);
        newThread.Start("The answer.");
    }

    public static void DoWork(object data)
    {
        Console.WriteLine("Static thread procedure. Data='{0}'", data);
    }

    public void DoMoreWork(object data)
    {
        Console.WriteLine("Instance thread procedure. Data='{0}'", data);
    }
}
```

Управляемый поток может быть фоновым и основным. Фоновый поток отличается от основного потока тем, что он не сохраняет управляемую среду выполнения в активном состоянии. Если в управляемом процессе (управляемой сборкой является EXE-файл) были остановлены все основные потоки, система останавливает все фоновые потоки и завершает работу. Для определения того, каким является поток (фоновым или основным) и для изменения его статуса используется свойство Thread.IsBackground. Поток можно сделать фоновым в любой момент, присвоив его свойству IsBackground значение true. Потоки, которые относятся к пулу управляемых потоков (то есть потоки, свойство IsThreadPoolThread которых имеет значение true), являются фоновыми. Все потоки, которые взаимодействуют с управляемой средой выполнения из неуправляемого кода, являются фоновыми. Все потоки, образованные путем создания и запуска нового объекта Thread, по умолчанию являются основными.

Платформа .NET Framework предоставляет примитивы синхронизации

для управления взаимодействием потоков и во избежание состояния гонки. Блокировки предоставляют единовременное управление ресурсом одному потоку или определенному количеству потоков. Поток, запрашивающий монопольную блокировку, если блокировка уже используется, блокируется до того момента, как блокировка станет доступной. Самым простым способом блокировки является операция lock, который управляет доступом к блоку кода:

```
class Program
{
    static readonly object _object = new object();

    static void A()
    {
        lock (_object)
        {
            DateTime time = DateTime.Now;
            Console.WriteLine("{0}:{1}", time, time.Millisecond);
            Thread.Sleep(100);
        }
    }

    static void Main()
    {
        for (int i = 0; i < 10; i++)
        {
            ThreadStart start = new ThreadStart(A);
            new Thread(start).Start();
        }
        Console.ReadKey();
    }
}
```

В этом примере каждый из 10 созданных потоков выполняют один и тот же метод, но будут выполнять его по очереди.

В .NET Framework имеется богатый набор других конструкций синхронизации, и они доступны в виде классов EventWaitHandle, Mutex и Semaphore. Некоторые из них практичнее других: Mutex, например, по большей части дублирует возможности lock, в то время как EventWaitHandle предоставляет уникальные возможности сигнализации. Все три класса основаны на абстрактном классе WaitHandle, но весьма отличаются по поведению. Одна из общих особенностей – это способность именования, делающая возможной работу с потоками не только одного, но и разных процессов. EventWaitHandle имеет два производных класса – AutoResetEvent и ManualResetEvent (не имеющие никакого отношения к событиям и делегатам

C#). Обоим классам доступны все функциональные возможности базового класса, единственное отличие состоит в вызове конструктора базового класса с разными параметрами. AutoResetEvent – наиболее часто используемый WaitHandle-класс и основная конструкция синхронизации, наряду с lock.

Предположим, требуется исполнять задачи в фоновом режиме без затрат на создание каждый раз нового потока для новой задачи. Этой цели можно достигнуть, используя единственный рабочий поток с постоянным циклом, ожидающим появление задачи. Получив задачу, он приступает к ее выполнению. После окончания выполнения поток снова переходит в режим ожидания. Это обычный многопоточный сценарий. Наряду с избавлением от накладных расходов на создание потоков получаем последовательно исполняющиеся задачи, устраняя потенциальные проблемы взаимодействия между потоками и чрезмерное потребление ресурсов.

Однако нужно решить что делать, если рабочий поток еще занят исполнением предыдущей задачи, а уже появилась следующая. Можно, например, блокировать исполнение, пока не завершена предыдущая задача. Это можно реализовать, используя два объекта типа AutoResetEvent – ready, который открывается (устанавливается путем вызова метода Set) рабочим потоком, когда он готов к работе, и go, который открывается вызывающим потоком, когда появляется новая задача. В следующем примере для демонстрации задачи используется простое строковое поле (объявленное с ключевым словом volatile для гарантии того, что оба потока будут видеть его в одном и том же состоянии):

```
class AcknowledgedWaitHandle
{
    static EventWaitHandle ready = new AutoResetEvent(false);
    static EventWaitHandle go = new AutoResetEvent(false);
    static volatile string task;

    static void Main()
    {
        new Thread(Work).Start();

        for (int i = 1; i <= 5; i++)
        {
            ready.WaitOne();
```

```
            task = "a".PadRight(i, 'h');
            go.Set();
        }

        ready.WaitOne();
        task = null;
        go.Set();
    }

    static void Work()
    {
        while (true)
        {
            ready.Set();
            go.WaitOne();

            if (task == null)
                return;

            Console.WriteLine(task);
        }
    }
}
```

ManualResetEvent позволяет потокам взаимодействовать друг с другом путем передачи сигналов. Обычно это взаимодействие касается задачи, которую один поток должен завершить до того, как другой продолжит работу. Когда поток начинает работу, которая должна быть завершена до продолжения работы других потоков, он вызывает метод Reset для того, чтобы поместить ManualResetEvent в несигнальное состояние. Этот поток можно понимать как контролирующий ManualResetEvent. Потоки, которые вызывают метод WaitOne в ManualResetEvent, будут заблокированы, ожидая сигнала. Когда контролирующий поток завершит работу, он вызовет метод Set для сообщения о том, что ожидающие потоки могут продолжить работу. Все ожидающие потоки освобождаются. Когда это было сообщено, ManualResetEvent остается в сигнальном состоянии до того момента, как оно будет снова установлено вручную. То есть, вызовы к WaitOne немедленно возвращаются. Можно контролировать начальное состояние ManualResetEvent, передав конструктору логическое значение, значение true, если начальное состояние сигнальное, и false, в противном случае.

Обращаться к элементам управления пользовательского интерфейса можно только из того потока, который создал этот элемент управления. Такое поведение обусловлено деталями реализации, в частности использованием

однопоточной модели (Single-threaded Apartment, STA) и механизма обработки оконных сообщений. В примере поток генерирует 1000 точек, которые добавляются в компонент график по очереди. Также прогресс выполнения индицируется компонентом ProgressBar.

```
delegate void InvokeDelegate(int newPoint, int progress);
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void startButton_Click(object sender, EventArgs e)
    {
        Thread t = new Thread(new ThreadStart(calcPoint));
        t.IsBackground = true;
        t.Start();
    }

    private void calcPoint()
    {
        Random rnd = new Random();
        int count = 1000;
        while (count-- > 0)
        {
            int newPoint = rnd.Next(-100, 100);
            int progress = (100 * (1000 - count)) / 1000;
            this.Invoke(new InvokeDelegate(updateChart), newPoint, progress);
            Thread.Sleep(1);
        }
    }

    private void updateChart(int newPoint, int progress)
    {
        progressBar.Value = progress;
        chart.Series[0].Points.AddY(newPoint);
    }
}
```

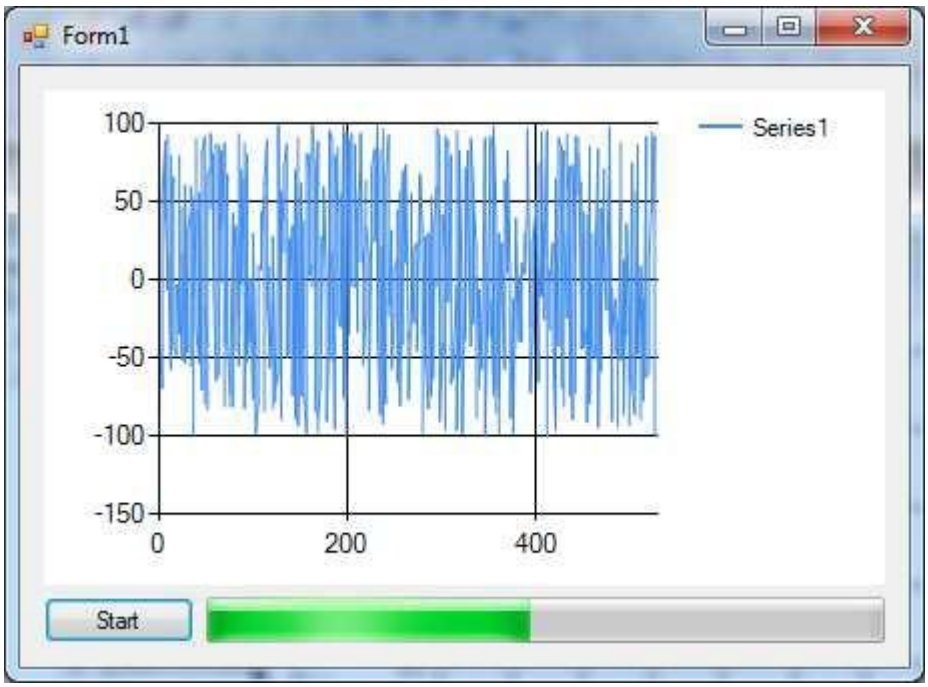

Рисунок 1 – Результат работы кода

Задания к лабораторной работе № 9

1) Напишите класс, позволяющий перемножать матрицы в нескольких потоках. Количество потоков должно задаваться в конструкторе класса.

2) Написать класс, реализующий многопоточный поиск матрицы с максимальной суммой элементов в матрице размером N×M. Результатом должны быть координаты массива в исходном массиве.

3) Написать компонент для построения графиков, позволяющий добавлять данные из разных потоков.

4**) Реализовать прямое и обратное дискретное преобразование Фурье в нескольких потоках. На вход функции для преобразования Фурье для теста подавать значения функции вида $y = A_1 * sin(w_1 * x) + A_2 * sin(w_2 * x) + ... + A_6 * sin(w_6 * x)$; где x ∈ [0; 2π], A_i , w_i – амплитуды и частоты сигналов входящих в итоговый сигнал y. Так же, для наглядной демонстрации предусмотреть возможность вывода этого графика на экран.

Контрольные вопросы

1) Что такое многопоточность?

2) Какие средства для работы с потоками имеются в C#

3) Опишите процесс создания и запуска потока.

4) Как остановить выполнение основного потока до тех пор, пока не завершат работу дочерние потоки?

5) Какие средства синхронизации потоков имеются в платформе .NET?

6) Как вывести данные из потока, напрямую в элемент управления расположенный на форме?