Потоки. Асинхронное и параллельное программирование

Формы параллельных вычислений

- Конкурентное исполнение (concurrency) - это наиболее общий термин, который говорит, что одновременно выполняется более одной задачи.
- Параллельное исполнение (parallel execution) подразумевает наличие более одного вычислительного устройства (например, процессора), которые будут *одновременно* выполнять несколько задач.

Формы параллельных вычислений

- Многопоточное исполнение (multithreading) - это один из способов реализации конкурентного исполнения путем выделения абстракции "рабочего потока" (worker thread).
- Асинхронное исполнение (asynchrony) подразумевает, что операция может быть выполнена кем-то на стороне: удаленным веб-узлом, сервером или другим устройством за пределами текущего вычислительного устройства.

Многопоточность

- ***Поток (thread)*** представляет собой независимую последовательность инструкций в программе.
- Многопоточность подразумевает использование множества потоков для обработки данных.

Задачи многопоточности

- Выполнение долгой задачи в отдельном потоке;
- Освобождение интерфейса на время выполнения задачи.

Пространство имен **System.Threading**

- Пространство имен System.Threading содержит классы и интерфейсы, которые дают возможность программировать в многопоточном режиме.

Отдельный поток - класс **Thread**

Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:

- Статическое свойство **CurrentThread** возвращает ссылку на выполняемый поток
- Свойство **IsAlive** указывает, работает ли поток в текущий момент
- Свойство **IsBackground** указывает, является ли поток фоновым
- Свойство **Name** содержит имя потока
- Свойство **Priority** хранит приоритет потока - значение перечисления ThreadPriority
- Свойство **ThreadState** возвращает состояние потока - одно из значений перечисления ThreadState

Отдельный поток - класс **Thread**

Методы класса Thread:

- Статический метод **Sleep** останавливает поток на определенное количество миллисекунд
- Метод **Abort** уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно.
  Метод **Join** блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
- Метод **Resume** возобновляет работу ранее приостановленного потока
- Метод **Start** запускает поток

Получение информации о потоке

```
// получаем текущий поток
Thread t = Thread.CurrentThread;

//получаем имя потока
Console.WriteLine("Имя потока: {0}", t.Name);
t.Name = "Метод Main";
Console.WriteLine("Имя потока: {0}", t.Name);

Console.WriteLine("Запущен ли поток: {0}", t.IsAlive)
Console.WriteLine("Приоритет потока: {0}", t.Priority
Console.WriteLine("Статус потока: {0}", t.ThreadState
```

```
Имя потока:
Имя потока: Метод Main
Запущен ли поток: True
Приоритет потока: Normal
Статус потока: Running
```

Статус потока - перечислении ThreadState

- **Aborted**: поток остановлен, но пока еще окончательно не завершен
- **AbortRequested**: для потока вызван метод Abort, но остановка потока еще не произошла
- **Background**: поток выполняется в фоновом режиме
- **Running**: поток запущен и работает (не приостановлен)
- **Stopped**: поток завершен
- **StopRequested**: поток получил запрос на остановку
- **Suspended**: поток приостановлен
- **SuspendRequested**: поток получил запрос на приостановку
- **Unstarted**: поток еще не был запущен
- **WaitSleepJoin**: поток заблокирован в результате действия методов Sleep или Join

Работа с потоком

```
    // создаем новый поток
    Thread myThread = new Thread(new ThreadStart(Count));
    myThread.Start(); // запускаем поток

    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("Главный поток:");
        Console.WriteLine(i * i);
        Thread.Sleep(300);
    }

    Console.ReadLine();
}

public static void Count()
{
    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("Второй поток:");
        Console.WriteLine(i * i);
        Thread.Sleep(400);
```

Работа с параметрами

```
int number = 4;
// создаем новый поток
Thread myThread = new Thread(new ParameterizedThreadStart(Count));
myThread.Start(number);

for (int i = 1; i < 9; i++)
{
    Console.WriteLine("Главный поток:");
    Console.WriteLine(i * i);
    Thread.Sleep(300);
}
```

```
public static void Count(object x)
{
    for (int i = 1; i < 9; i++)
    {
        int n = (int)x;

        Console.WriteLine("Второй поток:");
        Console.WriteLine(i*n);
        Thread.Sleep(400);
    }
}
```

Синхронизация потоков

- Для синхронизации используется ключевое слово **lock**.
  Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока.

```
static int x=0;
static object locker = new object();
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++)
    {
        Thread myThread = new Thread(Count);
        myThread.Name = "Поток " + i.ToString();
        myThread.Start();
    }

    Console.ReadLine()
}
```

```
public static void Count()
{
    lock (locker)
    {
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
            x++;
            Thread.Sleep(100);
        }
    }
}
```

Пространство имен **System.Threading.Tasks**

- Данная библиотека позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах, если на целевом компьютере имеется несколько ядер.

Класс **Task**

- Данный класс описывает отдельную задачу, которая запускается в отдельном потоке.
- Класс Task в качестве параметра принимает делегат **Action**. Этот делегат имеет определение public delegate void Action() .

```
static void Main(string[] args)
{
    Task task = new Task(Display);

    task.Start();
    Console.WriteLine("Выполняется работа метода Main");

    Console.ReadLine();
}
```

```
static void Display()
{
    Console.WriteLine("Начало работы метода Display");
    // имитация работы метода
    Thread.Sleep(3000);

    Console.WriteLine("Завершение работы метода Display");
}
```

Ожидание выполнения

- Если необходимо дождаться выполнения задачи, то используют метод – Wait().

```
static void Main(string[] args)
{
    Task task = new Task(Display);
    task.Start();
    Console.WriteLine("Выполняется работа метода Main");
    task.Wait();
    Console.ReadLine();
}
```

Работа с класом Task

- Конструктор класса Task принимает в качестве параметра делегат **Action** или **Action<object>.**

Свойства:

- **AsyncState**: возвращает объект состояния задачи
- **CurrentId**: возвращает идентификатор текущей задачи
- **Exception**: возвращает объект исключения, возникшего при выполнении задачи
- **Status**: возвращает статус задачи

```
Task task1 = new Task(()=>DisplayMessage("вызов метода с параметрами"));
task1.Start();

Task task2 = new Task(Display);
task2.Start();

Task task3 = new Task(() =>
{
    Console.WriteLine("Id задачи: {0}", Task.CurrentId);
});
task3.Start();

Task task4 = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Id задачи: {0}", Task.CurrentId);
});

TaskFactory tf = new TaskFactory();
Task t5 = tf.StartNew(Display);

Console.ReadLine();
```

Возвращение результата

```
Task<int> task1 = new Task<int>(()=>Factorial(5));
task1.Start();

Console.WriteLine("Факториал числа 5 равен {0}", task1.Result);
```

Асинхронное программирование

- Асинхронность позволяет вынести отдельные задачи из основного потока с специальные асинхронные методы или блоки кода. Особенно это актуально в графических программах, где продолжительные задачи могу блокировать интерфейс пользователя.

Асинхронный вызов

- Чтобы вызвать делегат в асинхронном режиме необходимо использовать метод **BeginInvoke/EndInvoke**.

```
public delegate int DisplayHandler();
static void Main(string[] args)
{
    DisplayHandler handler = new DisplayHandler(Display);

    IAsyncResult resultObj = handler.BeginInvoke(null, null);

    Console.WriteLine("Продолжается работа метода Main");
    int result = handler.EndInvoke(resultObj);
    Console.WriteLine("Результат равен {0}", result);

    Console.ReadLine();
}
```

BeginInvoke

Параметры:

- Параметры метода вызываемого делегата.
- Делегат System.AsyncCallback. AsyncCallback указывает на метод, который будет выполняться в результате завершения работы асинхронного делегата.
- Второй параметр представляет произвольный объект, с помощью которого мы можем передать дополнительную информацию в метод завершения.

```
static void Main(string[] args)
{
    DisplayHandler handler = new DisplayHandler(Display);

    IAsyncResult resultObj = handler.BeginInvoke(10, new AsyncCallback(AsyncCompleted), "Асинхронные вызовы");

    Console.WriteLine("Продолжается работа метода Main");

    int res = handler.EndInvoke(resultObj);

    Console.WriteLine("Результат: {0}", res);

    Console.ReadLine();
}
```

```
static int Display(int k)
{
    Console.WriteLine("Начинается работа метода Display....");

    int result = 0;
    for (int i = 1; i < 10; i++)
    {
        result += k * i;
    }
    Thread.Sleep(3000);
    Console.WriteLine("Завершается работа метода Display....");
   return result;
}

static void AsyncCompleted(IAsyncResult resObj)
{
    string mes = (string)resObj.AsyncState;
    Console.WriteLine(mes);
    Console.WriteLine("Работа асинхронного делегата завершена");
}
```

Ключевые слова **async и await**

- В .NET 4.5 во фреймворк были добавлены два новых ключевых слова async и await, цель которых - упростить написание асинхронного кода.
- Ключевое слово **async** указывает, что метод или лямбда-выражение может выполняться асинхронно. А оператор **await** позволяет остановить текущий метод, пока не завершится работа метода, помеченного как async, не останавливая выполнение потока.

```
static Task<int> FactorialAsync(int x)
{
    int result = 1;

    return Task.Run(() =>
    {
        for (int i = 1; i <= x; i++)
        {
            result *= i;
        }
        return result;
    });
}
```

```
static void Main(string[] args)
{
    DisplayResultAsync();
    Console.ReadLine();
}

static async void DisplayResultAsync()
{
    int num = 5;

    int result = await FactorialAsync(num);
    Thread.Sleep(3000);
    Console.WriteLine("Факториал числа {0} равен {1}", num, result);
}
```