

ASP.NET MVC и углубление изучения C#

Введение в мультипоточность



На этом уроке

1. Изучим понятие мультипоточности
2. Узнаем как запускать новые потоки
3. Познакомимся с одним из методов синхронизации
4. Узнаем в лицо деда Дедлока
5. Научимся обновлять пользовательский интерфейс в десктопном приложении

Оглавление

[На этом уроке](#)

[Введение в мультипоточность](#)

[Первые шаги в мультипоточности](#)

[Работа с потоками из редактора](#)

[Ограничения в потоках](#)

[Стандартная синхронизация на базе lock](#)

[Дед “deadlock”, который приносит боль и страдание](#)

[Дополнительная информация о взятии блокировок потоками](#)

[Работа с потоками в десктопных приложениях](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемые источники](#)

Введение в мультипоточность

Представьте, что вы пришли в большой супермаркет, где много прилавков с разными товарами. Но при этом в нем работает всего один продавец-кассир. В час пик этот магазин заполняется покупателями, а у касс выстраиваются огромные очереди. Конечно в реальной жизни такое встречается редко. Однако этот пример хорошо иллюстрирует идею мультипоточности, где поток выполнения является продавцом-кассиром, покупатели задачами, а супермаркет вашим приложением.

Разберемся с терминологией.

Процесс — это контейнер для потоков выполнения. Когда процесс начинает работу, он создает главный поток выполнения. Именно в нем происходит взаимодействие с пользовательским интерфейсом. При этом не важно, какой это процесс: консоль, приложение WPF или веб-сервис.

Поток — это наименьшая и дорогостоящая единица исполнения задач. Временем отработки потока выполнения на процессоре руководит именно она, а точнее, ее система планирования — планировщик.

Планировщик задач, строгий, но справедливый. Он старается выделить каждому потоку выполнения свой равный квант времени на процессоре.

Один из самых наглядных примеров работы с потоками — Microsoft Word. Пока вы печатаете текст, он автоматически проверяет орфографию и пунктуацию, а затем подчеркивает красным слова с ошибками. Если бы это все происходило в одном потоке, набор текста был бы очень затруднительным. Ведь Microsoft Word тогда должен был приостановить набора текста, проверить его на ошибки и только потом вернуть в работу.

Откройте диспетчер задач и перейдите на вкладку «Производительность». Выберите графики загрузки центрального процессора и посмотрите соотношения процессов к потокам.

Использование	Скорость	
2%	2,48 ГГц	
Процессы	Потоки	Дескрипторы
321	5056	157608

Процессов всегда будет меньше, чем потоков. Не переживайте из-за количества потоков, потому что операционная система гарантирует их исполнение.

Первые шаги в мультипоточности

Создайте простое консольное приложение. После этого мы создадим вручную новый поток, который будет работать параллельно главному, пока тот находится в состоянии «сна».

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Interview.Sample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Start");

            Thread thread = new Thread(new ThreadStart(SomeMethod));

            thread.Start();

            Thread.Sleep(9000);

            Console.WriteLine("Stop");
        }

        private static void SomeMethod()
        {
            for (int i = 0; i < 1000; i++)
            {
                Console.WriteLine($"{i}");

                Thread.Sleep(1000);
            }
        }
    }
}
```

Исполнение кода будет таким:

```
Start
0
1
2
3
4
5
6
7
8
Stop
9
10
11
12
13
14
15
16
17
18
...и так до 999
```

Давайте разберем каждую новую строчку кода:

1. Приложение стартует и создает один поток выполнения;
2. На консоль выводится слово "Start";
3. Затем код резервирует новый поток под переменной "thread" и в конструкторе передает метод, который должен работать новом потоке. Однако пока у нас есть только один поток выполнения, новый еще не запущен;
4. Запуск нового потока с помощью метода "Start()" у переменной "thread". Теперь у нас есть два потока, которые работают параллельно;
5. После запуска нового потока главный поток «засыпает» на 9 секунд, используя статический метод "Sleep(9000)" у статического класса "Thread";
6. "SomeMethod" работает параллельно методу "Main", выводит на консоль цифры и каждый раз после этого засыпает на 1 секунду;
7. Примерно через 9 секунд просыпается главный поток и выводит слово "Stop" на консоль;
8. Второй поток продолжает работу.

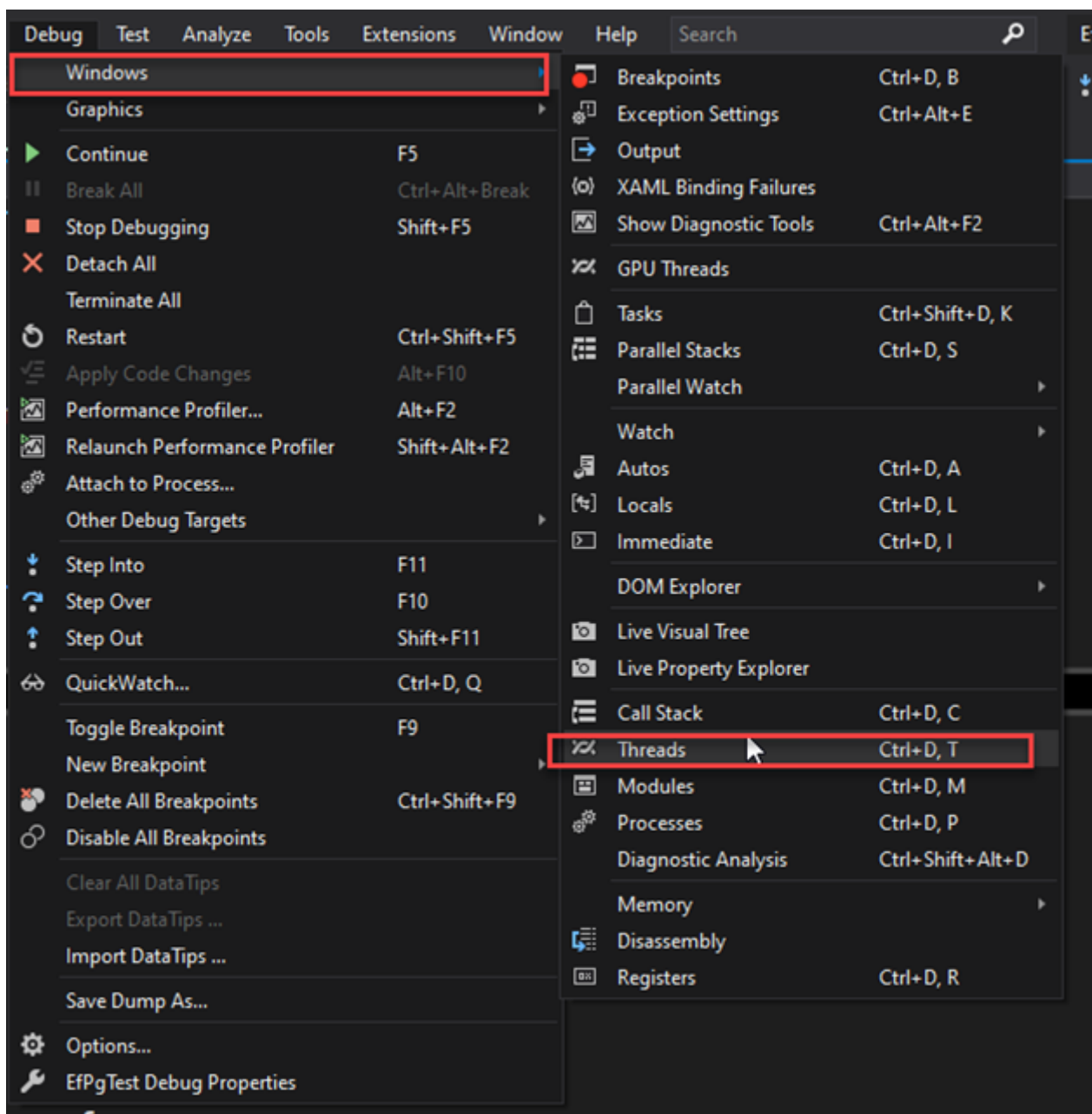
Мы рекомендуем вам поставить брейкпоинт в оба метода и посмотреть выполнение кода уже под отладкой.

Это очень простой, но важный пример. Он практически ничего не делает, но показывает, как работает идея мультиточечности.

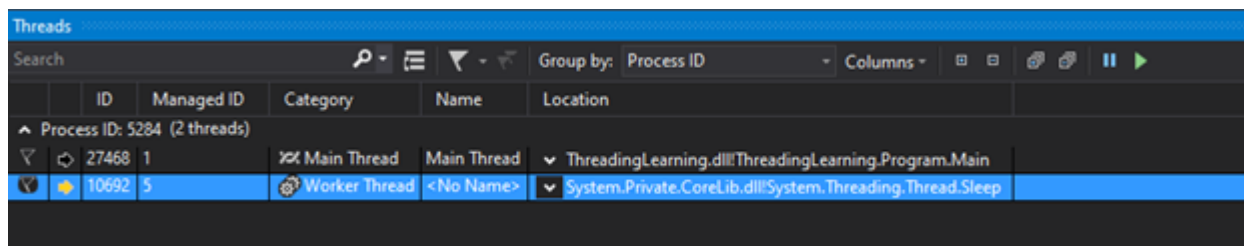
Работа с потоками из редактора

Почти любой редактор кода позволяет комфортно работать с потоками выполнения в отладке. Запустите предыдущее приложение под отладкой в Visual Studio, с установленным брейкпойнтом на строке "Thread.Sleep(9000);". Когда он сработает, включите дочернее окно отображения потоков "Threads", которое находится в меню "Debug->Windows->Threads".

Обратите внимание! Это меню можно увидеть только при запущенной отладке. В обычном режиме редактирования активировать окно "Threads" будет невозможно.



После активации окна вы увидите список ваших потоков:



The screenshot shows the 'Threads' window in Visual Studio. It has a search bar at the top, followed by icons for search, expand, filter, and zoom. Below these is a 'Group by' dropdown set to 'Process ID' and a 'Columns' dropdown. The table below has columns: ID, Managed ID, Category, Name, and Location. It shows two threads for Process ID 5284: a 'Main Thread' (Managed ID 1) and a 'Worker Thread' (Managed ID 5). The 'Worker Thread' is highlighted with a yellow arrow icon in the 'ID' column, indicating it is the current thread being debugged. The 'Name' column for the Worker Thread shows '<No Name>' and the 'Location' column shows 'System.Private.CoreLib.dll: System.Threading.Thread.Sleep'.

ID	Managed ID	Category	Name	Location
27468	1	Main Thread	Main Thread	ThreadingLearning.dll: ThreadingLearning.Program.Main
10692	5	Worker Thread	<No Name>	System.Private.CoreLib.dll: System.Threading.Thread.Sleep

Желтая стрелочка показывает, где сейчас находится отладка. А по двойному щелчку на элементе списка вы сможете перемещаться из кода одного потока в код другого. Это очень важный функционал, которым не стоит пренебрегать при написании мультипоточного кода.

Ограничения в потоках

Нельзя обновлять данные из разных потоков, иначе будет полный хаос. Если все же попытаться это сделать, то в лучшем случае вы получите ошибку, а в худшем — неправильные данные и поломку логики.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Interview.Sample
{
    class Program
    {
        private static int count = 1;

        static void Main(string[] args)
        {
            Console.WriteLine("Start");

            for (int i = 0; i < 5; i++)
            {
                Thread thread = new Thread(new ThreadStart(() =>
CountMethod(i)));

                thread.Start();

                Thread.Sleep(1000);

                Console.WriteLine("Stop");
            }

            private static void CountMethod(int threadId)
            {
```

```

        count = 1;

        for (int i = 0; i < 1000; i++)
        {
            Console.WriteLine($"{count} in {threadId}");

            count = count + 1;

            Thread.Sleep(1000);
        }
    }
}

```

Этот пример показывает, что на консоль будет выводиться хаотичный подсчет.

К примеру, вы получите такой вывод:

```

Start
1 in 1
1 in 2
1 in 3
1 in 4
1 in 5
3 in 2
3 in 1
3 in 3
6 in 4
6 in 5
Stop
8 in 3
8 in 1
8 in 2
11 in 4
11 in 5
13 in 1
13 in 2
13 in 3
16 in 5
16 in 4
18 in 2
18 in 3
18 in 1

```

Так происходит, потому что каждый запущенный поток обновляет переменную count при старте в 1 и затем каждую секунду увеличивает на единицу. Соответственно, обновление этой переменной происходит для всех потоков сразу, а вывод на консоли является неверным по отношению к исполняющему потоку. Вспомните наш супермаркет и бедного кассира, который хаотично берет товар и оплату у разных покупателей.

Чтобы избежать таких хаотичных модификаций, есть методы синхронизации потоков. Самый распространенный — оператор “lock”, за которым скрывается очень сложная логика.

Стандартная синхронизация на базе lock

Давайте модифицируем предыдущий пример с оператором lock:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Interview.Sample
{
    class Program
    {
        private static int count = 1;

        private static object lockObject = new object();

        static void Main(string[] args)
        {
            Console.WriteLine("Start");

            for (int i = 0; i < 5; i++)
            {
                Thread thread = new Thread(new ThreadStart(() =>
CountMethod(i)));

                thread.Start();

                Thread.Sleep(1000);

                Console.WriteLine("Stop");
            }

            private static void CountMethod(int threadId)
            {
                lock (lockObject)
                {
                    count = 1;

                    for (int i = 0; i < 10; i++)
                    {
                        Console.WriteLine($"{count} in {threadId}");

                        count = count + 1;

                        Thread.Sleep(1000);
                    }
                }
            }
        }
    }
}
```

Нужно ввести некий пустой объект (в нашем случае это object) с названием переменной "lockObject". Затем необходимо в методе исполнения потока завернуть его в конструкцию "lock(..)". Синтаксис будет как при использовании оператора "try { } catch(Exception ...)".

Запустите пример и убедитесь, что вызов происходит поочередно:

```
Start
1 in 1
Stop
2 in 1
3 in 1
4 in 1
5 in 1
6 in 1
7 in 1
8 in 1
9 in 1
10 in 1
1 in 2
2 in 2
3 in 2
4 in 2
5 in 2
6 in 2
7 in 2
```

Этот оператор блокирует переменную "lockObject" и держит ее до тех пор, пока поток не отпустит, то есть не выйдет за скобки видимости. После этого другие потоки могут заблокировать этот объект. Если другие потоки пытаются взять блокировку над объектом, который уже был заблокирован, они останавливаются и ждут её снятия. Оператор "lock" гарантирует эксклюзивную блокировку объекта на один поток.

Одна из самых страшных и сложно диагностируемых проблем в разработке программного обеспечения — "dead lock" или "дедлок".

“deadlock”, который приносит боль и страдание

Почти в каждом языке программирования есть свой механизм ловли непредвиденных ситуаций. Например, в C# этим управляют оператор "try-catch", который может поймать ошибку, и предотвратить переход программы из нестабильного состояния в рабочее. Однако есть логические ошибки, которые

не сможет поймать никакой оператор. Одна из самых распространенных — “dead lock”. И вот как он выглядит:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Interview.Sample
{
    class Program
    {
        private static object lockObjectOne = new object();
        private static object lockObjectTwo = new object();

        static void Main(string[] args)
        {
            Console.WriteLine("Start");

            Thread threadOne = new Thread(new
ThreadStart(ThreadOneMethod));
            Thread threadTwo = new Thread(new
ThreadStart(ThreadTwoMethod));

            threadOne.Start();
            threadTwo.Start();

            Thread.Sleep(1000);

            Console.WriteLine("Stop");
        }

        private static void ThreadOneMethod()
        {
            lock (lockObjectOne)
            {
                Thread.Sleep(1000);

                lock (lockObjectTwo)
                {
                    Console.WriteLine("This will be never executed...");
                }
            }
        }

        private static void ThreadTwoMethod()
        {
            lock (lockObjectTwo)
            {
                Thread.Sleep(1000);

                lock (lockObjectOne)
                {
                    Console.WriteLine("And this will be never
executed...");
                }
            }
        }
    }
}
```

```
}  
}
```

Если вы запустите этот пример в редакторе, то при выполнении программы не увидите вывод на консоль текста после вторых операторов “lock”. Однако при этом программа будет работать и никогда не завершится, потому что:

1. Первый поток стартует и берет блокировку объекта “lockObjectOne”;
2. Затем засыпает на 1 секунду;
3. Второй поток стартует и берет блокировку объекта “lockObjectTwo”;
4. После этого он засыпает на 1 секунду;
5. Первый поток просыпается и пытается взять блокировку объекта “lockObjectTwo”. Но из-за того, что этот объект уже заблокирован вторым потоком, первый переходит в ожидание до тех пор, пока блокировка не будет снята;
6. Второй поток просыпается и пытается взять блокировку объекта “lockObjectOne”, который заблокирован первым потоком, а затем тоже переходит в состояние ожидания;
7. Оба потока ждут, когда блокировки будут сняты, но этого никогда не произойдет.

Они так и будут сидеть и ждать друг друга, то есть повиснут во взаимоблокировке.

На собеседованиях иногда спрашивают: «Есть ли такое понятие, как “DeadLockException” и можно ли его отловить?».

Ответ прост: «Такой ошибки нет и отловить в операторе “try-catch” эту ситуацию невозможно. Так как с точки зрения программы, ошибок нет – просто бесконечное ожидание».

В этом примере легко понять, где возникает взаимоблокировка, а затем исправить проблему. Но представьте себе большие комплексы, где классов больше, чем учеников в средней школе. Поэтому ошибку легче предотвратить, чем исправить. Иногда на это уходят целые недели. Поэтому продумывайте своё приложение на схемах, прежде чем писать код.

Дополнительная информация о взятии блокировок потоками

Также на собеседованиях могут спросить, что произойдет при выполнении вот такого примера:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace Interview.Sample
{
    class Program
    {
        private static object lockObjectOne = new object();

        static void Main(string[] args)
        {
            Console.WriteLine("Start");

            Thread thread = new Thread(new ThreadStart(ThreadOneMethod));

            thread.Start();

            Thread.Sleep(1000);

            Console.WriteLine("Stop");
        }

        private static void ThreadOneMethod()
        {
            lock (lockObjectOne)
            {
                Thread.Sleep(1000);

                lock (lockObjectOne)
                {
                    Console.WriteLine("What's happens here?");
                }
            }
        }
    }
}
```

Это попытка в одном и том же потоке взять блокировку на один объект. В таком случае ошибки не будет, потому что блокировка принадлежит этому потоку и попытка взять ее во второй раз не вызовет никаких проблем.

Работа с потоками в десктопных приложениях

Десктопные приложения требуют большего внимания к работе с потоками. При работе с ними нельзя обновлять пользовательский интерфейс не из главного пользовательского потока.

Необходимо работать с пользовательским интерфейсом из разных потоков через диспетчер. Он существует у каждого элемента управления: от обычного окошка до кнопки. И до него можно достучаться из статической ссылки.

Если не использовать диспетчер при обновлении окна WPF или WinForms, это приведет к ошибке, которая вызовет исключение `System.InvalidOperationException: 'Вызывающий поток не может получить доступ к данному объекту, так как владельцем этого объекта является другой поток.'`:

```

using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Windows.Threading;

namespace WpfThreading
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            Thread f = new Thread(Foo);

            f.Start();
        }

        public void Foo()
        {
            Thread.Sleep(2000);

            SimpleTextBox.Text = SimpleTextBox.Text + "Failed to update
from another thread!";
        }
    }
}

```


Перепишем код на корректный:

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Windows.Threading;

namespace WpfThreading
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            Thread f = new Thread(Foo);

            f.Start();
        }

        public void Foo()
        {
            Thread.Sleep(2000);

            Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background,
            new Action(() =>
            {
                SimpleTextBox.Text = SimpleTextBox.Text + "Updated from
another thread!";
            }));
        }
    }
}
```

Если запустить этот пример, то текстовое поле с названием "SimpleTextBox" на форме главного окна обновится. Это удобно, если вам нужно обновить большую таблицу новыми данными или проверить орфографию в большом тексте.

Благодаря использованию потоков ваше приложение становится более отзывчивым для пользователя.

Домашнее задание

1. Создайте форму WPF или WinForms, разместите на ней текстовое поле и в другом потоке последовательно добавляйте на него числа Фибоначчи.
2. В этой же форме добавьте регулятор, который будет отсчитывать, сколько секунд должно пройти, прежде чем появится следующее число.
3. Изучите внимательно статичный класс Thread и не статичный класс. Найдите метод, который может прервать выполняющийся поток и зафиксируйте ту ошибку, которая формируется при отмене.
4. Создайте класс-обертку над List<T>, что бы можно было добавлять и удалять элементы из разных потоков без ошибок.
5. *Вернитесь к вашему старому проекту по отображению графиков с агентов мониторинга. Сделайте новое приложение, которое забирает данные через потоки, не используя async/await.

Дополнительные материалы

1. <https://docs.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading>

Используемые источники

1. <https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=net-5.0>
2. <https://docs.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading>