

Введение В МНОГОПОТОЧНОСТЬ

ASP.NET MVC и углубление изучения C#

Что будет на уроке

1. Что такое многопоточность
2. Создание потока
3. Синхронизация
4. Ошибки || программирования
5. Средства для ||
программирования в C#

Разбираемся с ДЗ. Вопросы?

Как вы считаете, потокобезопасный ли у вас получился сервер (веб-приложение)?

Корректно ли он отрабатывает если одновременно прибегут много пользователей?

Как вы считаете, потокобезопасный ли у вас получился сервер (веб-приложение)?

Если нет, то Почему?

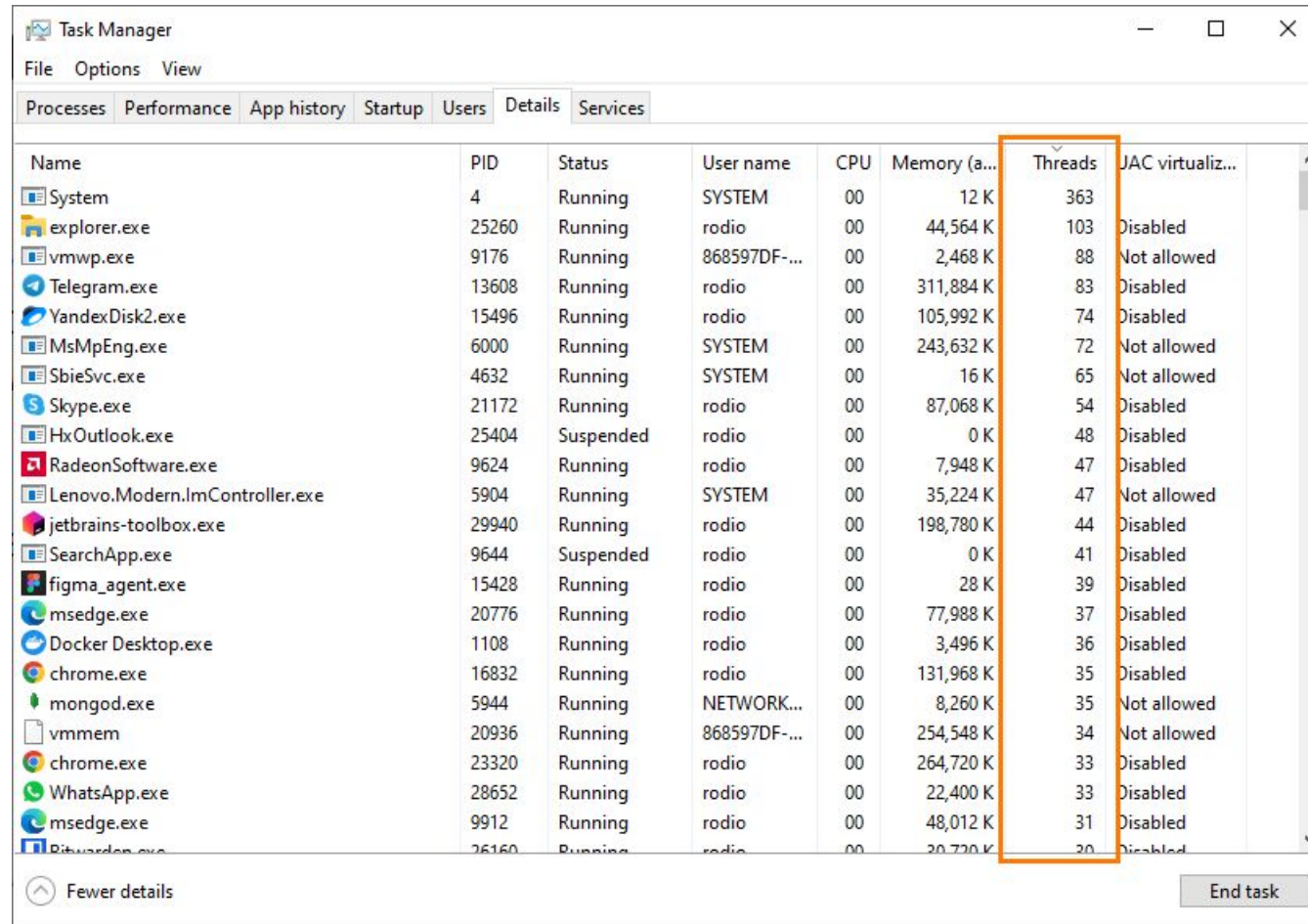
Процесс — контейнер для потоков. У него есть как минимум один поток исполнения.

Поток — наименьшая и дорогостоящая единица исполнения задач. Поток не может быть без процесса.

Процесс

- Для каждого процесса ОС выделяет защищенный участок памяти, к которому другие процессы не имеют доступа (хотя и есть нюансы: тыц и тыц)
- Потоки одного и того же процесса имеют доступ к памяти друг друга
- Процесс включает в себя минимум один поток
- Процессов может быть намного меньше, чем потоков

Количество потоков



Task Manager

File Options View

Processes Performance App history Startup Users Details Services

Name	PID	Status	User name	CPU	Memory (a...	Threads	JAC virtualiz...
System	4	Running	SYSTEM	00	12 K	363	
explorer.exe	25260	Running	rodio	00	44,564 K	103	Disabled
vmwp.exe	9176	Running	868597DF-...	00	2,468 K	88	Not allowed
Telegram.exe	13608	Running	rodio	00	311,884 K	83	Disabled
YandexDisk2.exe	15496	Running	rodio	00	105,992 K	74	Disabled
MsMpEng.exe	6000	Running	SYSTEM	00	243,632 K	72	Not allowed
SbieSvc.exe	4632	Running	SYSTEM	00	16 K	65	Not allowed
Skype.exe	21172	Running	rodio	00	87,068 K	54	Disabled
HxOutlook.exe	25404	Suspended	rodio	00	0 K	48	Disabled
RadeonSoftware.exe	9624	Running	rodio	00	7,948 K	47	Disabled
Lenovo.Modern.ImController.exe	5904	Running	SYSTEM	00	35,224 K	47	Not allowed
jetbrains-toolbox.exe	29940	Running	rodio	00	198,780 K	44	Disabled
SearchApp.exe	9644	Suspended	rodio	00	0 K	41	Disabled
figma_agent.exe	15428	Running	rodio	00	28 K	39	Disabled
msedge.exe	20776	Running	rodio	00	77,988 K	37	Disabled
Docker Desktop.exe	1108	Running	rodio	00	3,496 K	36	Disabled
chrome.exe	16832	Running	rodio	00	131,968 K	35	Disabled
mongod.exe	5944	Running	NETWORK...	00	8,260 K	35	Not allowed
vmmem	20936	Running	868597DF-...	00	254,548 K	34	Not allowed
chrome.exe	23320	Running	rodio	00	264,720 K	33	Disabled
WhatsApp.exe	28652	Running	rodio	00	22,400 K	33	Disabled
msedge.exe	9912	Running	rodio	00	48,012 K	31	Disabled
Pitrudd.exe	26160	Running	rodio	00	20,720 K	20	Disabled

^ Fewer details

End task

Поток

- Потоков исполнения может быть много
- Потоки могут быть в состоянии сна (например, при вызове `Thread.Sleep`)
- Каждый поток исполняется на процессоре определенный квант времени
- Распределение работы потоков исполнения регулируется операционной системой (планировщиком потоков)
- **Реальное кол-во одновременно выполняемых потоков ограничено кол-вом ядер процессора(ов)**
- Каждый поток имеет стек и потребляет ресурсы ОС
- А создание потока требует перехода в kernel space, что в итоге выходит довольно дорого

Что означает многопоточность?

Многопоточность – форма конкурентности, использующая несколько программных потоков выполнения.

Конкурентность – выполнение сразу нескольких действий в одно и то же время.

Класс Task

1. Представляет доступ к созданию высокоуровневых эффективных потоков на C#.
2. Имеет простой синтаксис (`Task.Run(() ⇒ работа...)`), а также позволяет легко дожждаться окончания выполнения (ключевое слово `await`).
3. За создание новых потоков в этом случае отвечает пул потоков (`ThreadPool`).

Создание и запуск задачи

```
Task.Run(() =>  
{  
    DoHeavyOperation();  
});
```

ThreadPool

- Когда код ставит работу в очередь пула потоков, то сам пул потоков в случае необходимости позаботится о создании потока
- Мы более не тратим время на создание потока ОС: мы работаем на уже созданных
- Либо ограничивает пропускную способность либо наоборот: даёт возможность работать на все 100% от всех процессорных ядер

Класс Thread

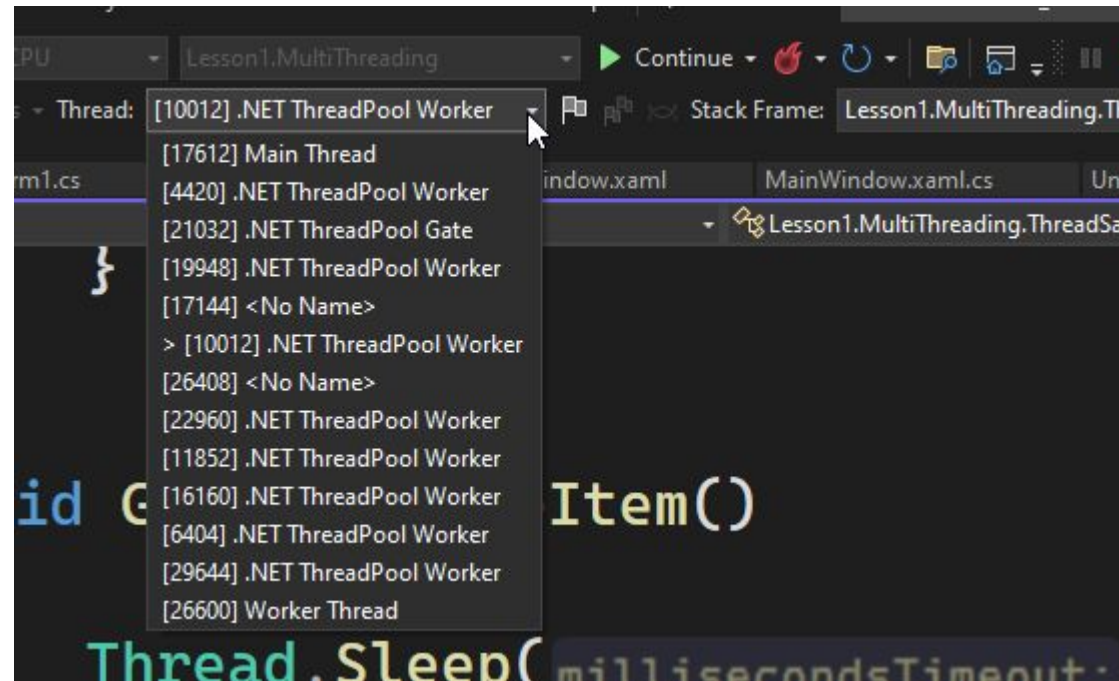
- Thread относится к низкоуровневым абстракциям, с его помощью создаются потоки “по старинке”
- Обычно потоки, созданные через Thread менее эффективны, чем потоки, созданные через высокоуровневые абстракции типа Task
- В современной разработке на C# почти не используется

Как только вы вводите команду `new Thread()`, все кончено: ваш проект уже содержит устаревший код.

© Stephen Cleary (“Конкурентность в C#”)

Отладка многопоточного кода в VS

В Visual Studio используйте список Thread для просмотра потоков и их отладки:



Многопоточность в ASP.NET Core

- Запросы в ASP.NET Core могут выполняться одновременно (на нескольких потоках процессора)
- Это увеличивает производительность сервера
- Но и создает проблемы. Всегда нужно держать в голове тот факт, что код может выполняться параллельно.

Вопросы

Синхронизация

примитивы синхронизации потоков

Примитив синхронизации (ПС)

- ПС можно сравнить с охранником, который следит за тем, чтобы в заданной области одновременно выполнялось не более N потоков (обычно 1)
- ПС, ограничивающие одновременное выполнение кода **одним** потоком называют монопольными (lock (Monitor), SpinLock, Mutex)
- ПС, позволяющие ограничить одновременное выполнение 2-мя и более потоками называют немонопольными (ReaderWriterLockSlim, SemaphoreSlim)

Монопольное блокирование

- ПС монопольного блокирования позволяют запускать определенный блок кода только одному потоку
- Такой блок кода называется критической секцией
- Например, в аэропорту критической секцией будет являться пункт досмотра, в котором в один момент времени может обслуживаться только один человек
- Поэтому такие секции часто становятся узким горлышком



Оператор lock

Самый распространенный и простой оператор для синхронизации потоков.

- Синтаксис схож с try-catch
- Для работы нужно вводить дополнительные ссылочные переменные, которые будут браться в блокировку
- Если блокировка уже была взята, то текущий поток засыпает (переходит в состояние `WaitSleepJoin`)
- Нельзя использовать структуры в качестве объектов блокировок

Оператор lock: Синтаксис

```
object syncObj = new();  
lock (syncObj) //syncObj - объект синхронизации  
{  
    /*  
    критическая секция, в которой код  
    будет выполняться только на одном потоке  
    */  
}
```

Объект синхронизации

Сейчас хорошим тоном является создание нового приватного объекта для синхронизации:

```
private object _syncObj = new object();
```

Пример с lock: Безопасный каталог

```
public class Catalog
{
    private List<Product> _products = new();
    private readonly object _syncObj = new();

    public void AddProduct(Product product)
    {
        lock (_syncObj)
        {
            _products.Add(product);
        }
    }

    public void RemoveProduct(Product product)
    {
        lock (_syncObj)
        {
            _products.Remove(product);
        }
    }
}
```


Задача

Надо ли в этом случае синхронизировать метод чтения товаров?

```
public List<Product> GetProducts()  
{  
    return _products;  
}
```

VS

```
public List<Product> GetProducts()  
{  
    lock (_syncObj)  
    {  
        return _products;  
    }  
}
```



Реализация метода Add у класса List

```
01 public void List.Add(T item)
02 {
03     _version++;
04     T[] array = _items;
05     int size = _size;
06     if ((uint)size < (uint)array.Length)
07     {
08         _size = size + 1;
09         array[size] = item;
10     }
11     else
12     {
13         AddWithResize(item);
14     }
15 }
```

Если 2-й поток между 8-й и 9-й строкой обратится к products, то последний элемент такого листа будет равен null. Такая ситуация называется [Data Race](#).

А что с `foreach` по листу?

На самом деле если во время перебора элементов лист изменится, то на следующем же шаге итерации выбросится исключение `InvalidOperationException` из-за несовпадения версии листа

** `LINQ` методов тоже касается*

Кстати, правильнее так

```
public IReadOnlyList<Product> GetProducts()  
{  
    lock (_sync)  
    {  
        return _products;  
    }  
}
```

Когда нужна синхронизация?

Нужна синхронизация:

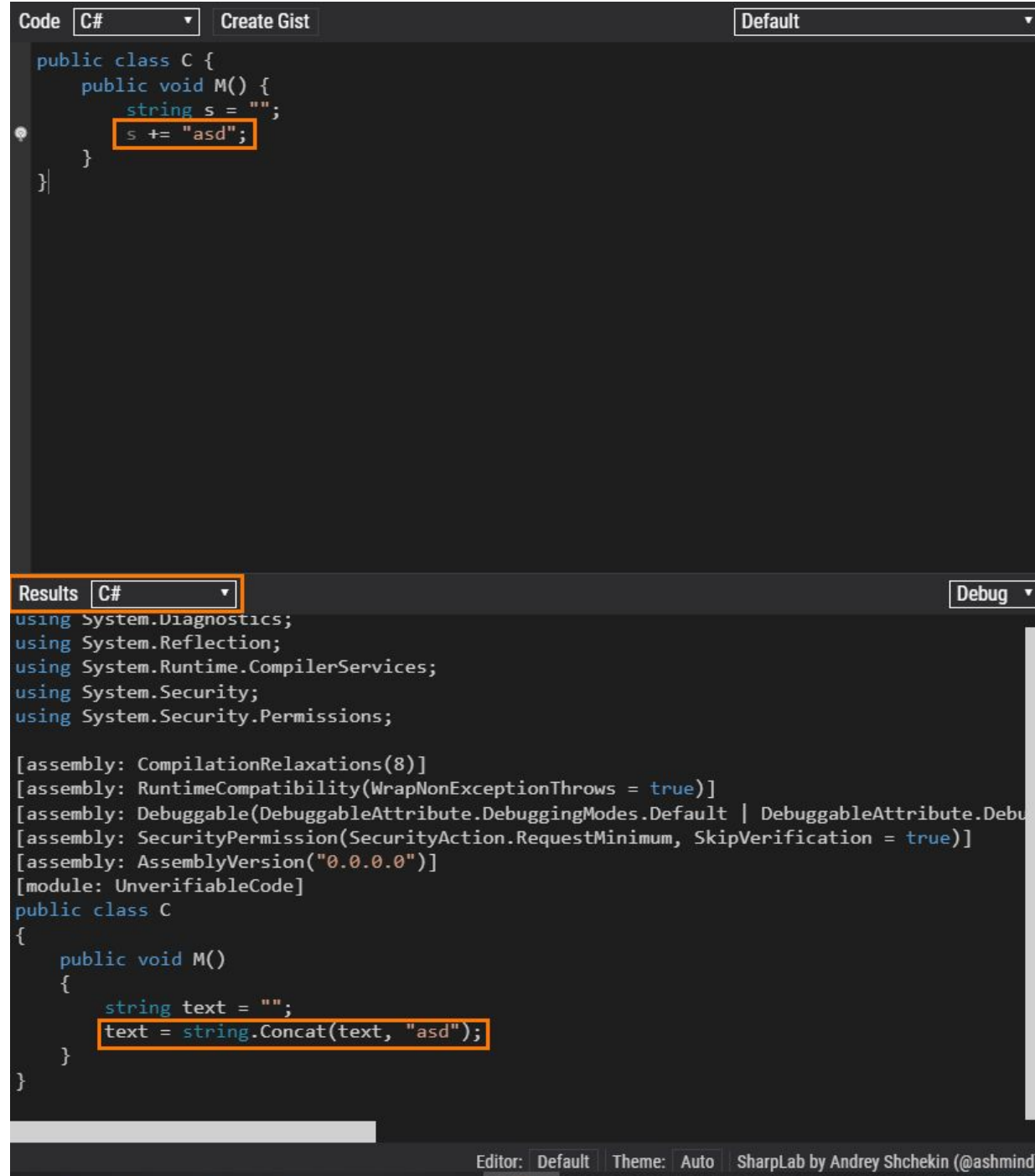
- Запись, Запись
- Запись, Чтение
- Только чтение (обычно нет)

**lock – всего лишь
синтаксический
сахар.**

Вот во что
компилятор
разворачивает его
на самом деле:

```
object syncObj = new();
bool lockTaken = false;
try
{
    Monitor.Enter(syncObj, ref lockTaken);
    /*
     критическая секция, в которой код
     будет выполняться
     только на одном потоке
    */
}
finally
{
    if (lockTaken)
    {
        Monitor.Exit(syncObj);
    }
}
```

Кстати, посмотреть
во что в итоге
компилятор
разворачивает
ваш код можно на
сайте sharplab.io



```
Code C# Create Gist Default

public class C {
    public void M() {
        string s = "";
        s += "asd";
    }
}

Results C# Debug

using System.Diagnostics;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Security;
using System.Security.Permissions;

[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.Default | DebuggableAttribute.DebuggingModes.IgnoreSymbolicExecutionDiagnostics)]
[assembly: SecurityPermission(SecurityAction.RequestMinimum, SkipVerification = true)]
[assembly: AssemblyVersion("0.0.0.0")]
[module: UnverifiableCode]
public class C
{
    public void M()
    {
        string text = "";
        text = string.Concat(text, "asd");
    }
}
```

Editor: Default Theme: Auto SharpLab by Andrey Shchekin (@ashmind)

Вопросы

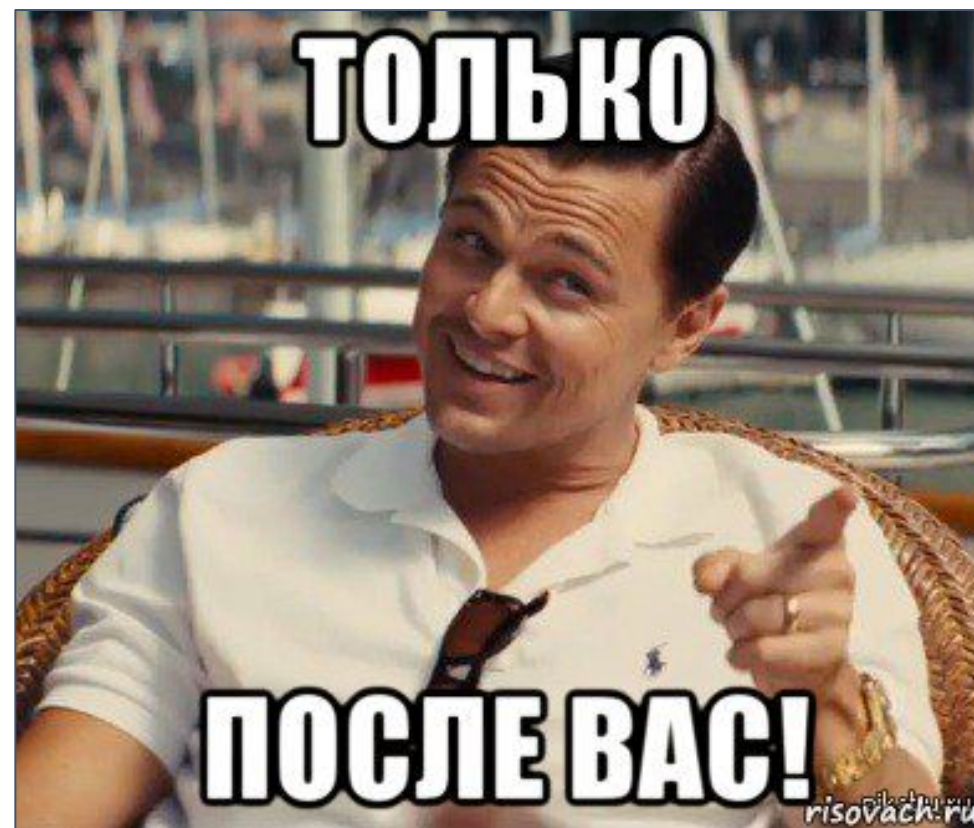
Немонопольное блокирование

ReaderWriterLockSlim

Ошибки || программирования

Deadlock или “После Вас”

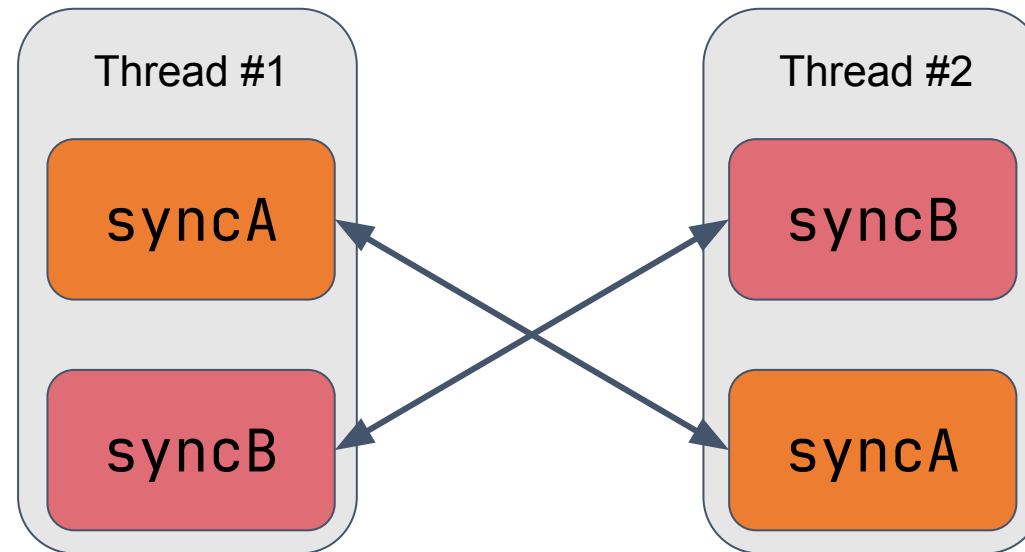
- Сложная логическая ошибка
- **Взаимоблокировка** двумя потоками объектов блокирования
- В случае дедлока ни один поток никогда больше не продолжит свое выполнение
- Не пробрасывается исключение, не ловится try-catch
- Диагностика таких ошибок довольно трудоемкая



Классический перекрестный дедлок

1-й поток захватывает ПС syncA, затем ПС syncB

2-й поток захватывает те же ПС в обратном порядке (syncB, syncA)



Пример перекрестного дедлока

Thread #1:

```
public void ClearCatalog()
{
    lock (_categories)
    lock (_products)
    {
        _categories.Clear();
        _products.Clear();
    }
}
```

Thread #2:

```
// Урезает названия до maxCount символов
public void ShrinkNames(int maxCount)
{
    lock (_products)
    lock (_categories)
    {
        // тут логика
    }
}
```

Deadlock: Решение

1. Захват примитивов синхронизации всегда в одинаковом порядке
2. Не захватывать более одного примитива синхронизации вовсе

Задача. Будет ли этот код работать?

```
object syncObj = new();  
lock (syncObj)  
{  
    // логика ...  
    lock (syncObj)  
    {  
        // еще логика ...  
    }  
}
```

Да, код будет работать корректно, т. к. Monitor относится к рекурсивным ПС и позволяет одному и тому же потоку брать блокировку сколько угодно раз.

Когда это может быть актуально?

Пример полезности рекурсивной блокировки

Data Race (гонка данных)

- Это состояние когда разные потоки обращаются к одной ячейке памяти без какой-либо синхронизации и как минимум один из потоков осуществляет запись.
- Параллельное программирование еще называют программированием между строк

Data Race

```
public void DoWork()  
{  
    i++;  
}
```

```
IL_0001: ldarg.0          // this  
IL_0002: ldflld          IL_0007: ldc.i4.1  
IL_0008: add  
IL_0009: stfld
```

value-typed объект синхронизации

- При попытке в качестве объекта блокировки передать значимый тип (value type), блок синхронизации просто не будет работать.
- Так происходит потому, что метод `Monitor.Enter` ожидает ссылочный тип и при передаче в него типа-значения происходит операция упаковки (boxing), которая всегда создает новый объект.

value-typed обект синхронизации

```
var num = 1;  
lock ((object) num)  
{  
    // code ...  
}
```

Shared обѐкт синхронизации

```
lock (this)  
{  
    // code ...  
}
```

```
lock (typeof(Product))  
{  
    // code ...  
}
```

Interlocked



TPL

Task Parallel Library

Parallel.For из TPL

```
Parallel.For(0, 1_000_000, new ParallelOptions()  
{  
    MaxDegreeOfParallelism = 3  
}, i =>  
{  
    list.Add(i);  
});
```

Про удаление из ConcurrentBag

Ошибки, связанные с асинхронным программированием

Их разберем чуть позже

Вызов асинхронных методов как синхронных

- При вызове асинхронного метода как синхронного в лучшем случае вы лишитесь преимуществ асинхронности, т. к. вызывающий поток заблокируется на время вызова асинхронного метода.
- В среде выполнения с контекстом синхронизации (WinForms, WPF, Unity) вся ваша программа войдет в дедлок и заблокируется навсегда (попросту говоря зависнет). [ссылка на пример]

Вызов асинхронных методов как синхронных в WPF: Deadlock

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    WaitOneSec().GetAwaiter().GetResult(); //дедлок случится здесь
    MessageBox.Show("Это сообщение не покажется никогда");
}
```

```
private async Task WaitOneSec()
    ⇒ await Task.Delay(TimeSpan.FromSeconds(1));
```



DEMO

Потокобезопасные коллекции

Итоги

- ПС замедляют программу, поэтому старайтесь избегать их использования. Альтернатива: возможности Interlocked и/или потокобезопасные коллекции
- Старайтесь самостоятельно не распараллеливать задачи при разработке бекенда на ASP.NET Core
- Перед тем, как добавить библиотеку в свою программу проверьте ее потокобезопасность и только после этого примите решение о времени жизни зависимости

Итоги

- Старайтесь мыслить параллельно и читайте между строк :)
 - Если сомневаетесь, преобразуйте код в IL или даже в ASM при помощи sharplab.io и перепроверьте, либо просто задайте вопрос создателю библиотеки (только, увы, они сами не всегда знают)
- Даже потокобезопасные классы (коллекции) гарантируют потокобезопасность только для своих методов. Поэтому не расслабляйтесь и всегда думайте о потокобезопасности.

Полезные материалы

[Курс “Параллельное программирование”](#) (Евгений Калишенко)

[Семинары CLRium “Concurrency и Parallelism”](#) (Стас Сидристый)

[Online-книга DotNetBook \(Threads\)](#) (Стас Сидристый)

Конкурентность в C# (Stephen Cleary)



CLR via C# (Джеффри Рихтер)

Часть V. Многопоточность
(стр. 723-892)



Домашнее задание

1. Сделайте класс `Catalog` потокобезопасным
2. Создайте собственный потокобезопасный класс `ConcurrentList<T>`. Чтобы можно было добавлять и **удалять** элементы из разных потоков без ошибок, а также очищать список.
 - 2.1 ★ Постарайтесь сделать свою коллекцию максимально быстрой
 - 2.2 ★★ Реализуйте возможность обхода вашего класса через цикл `foreach`
 - 2.3 ★★★ Добавьте потокобезопасный метод сортировки элементов в вашей коллекции

Домашнее задание

- Параллельное программирование сложная тема, требующая “параллельного мышления”
- Для того, чтобы его развить рекомендую порешать задачи на следующих слайдах
- Кстати, некоторые из них могут спросить на собеседовании

Реализация метода Add у класса List

```
01 public void Add(T item)
02 {
03     _version++;
04     T[] array = _items;
05     int size = _size;
06     if ((uint)size < (uint)array.Length)
07     {
08         _size = size + 1;
09         array[size] = item;
10     }
11     else
12     {
13         AddWithResize(item);
14     }
15 }
```

Задание: Разберитесь
Что конкретно делает операцию
добавления в List потокобезопасной?

Какие варианты МОЖЕТ ВЫВЕСТИ ЭТОТ КОД?

```
static int x;  
  
static void Run()  
{  
    var t1 = Task.Run(() =>  
    {  
        x = 1;  
        Console.Write(x);  
    });  
    var t2 = Task.Run(() =>  
    {  
        x = 0;  
        Console.Write(x);  
    });  
    t1.Wait();  
    t2.Wait();  
}
```

Какие варианты МОЖЕТ ВЫВЕСТИ ЭТОТ КОД?

```
static int x, y;

public static void Run()
{
    var t1 = Task.Run(() =>
    {
        x = 1;
        Console.WriteLine(y);
    });
    var t2 = Task.Run(() =>
    {
        y = 1;
        Console.WriteLine(x);
    });
    t1.Wait();
    t2.Wait();
}
```


Спасибо!
Вопросы.

Каждый день
вы становитесь
лучше :)

