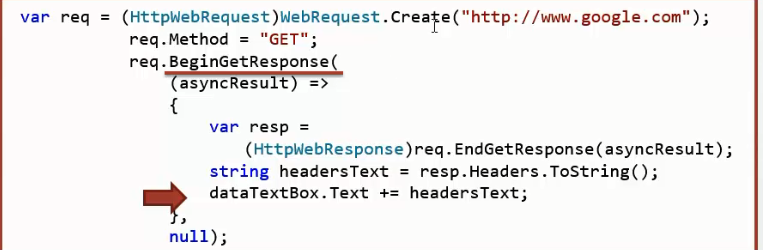


Синхронно

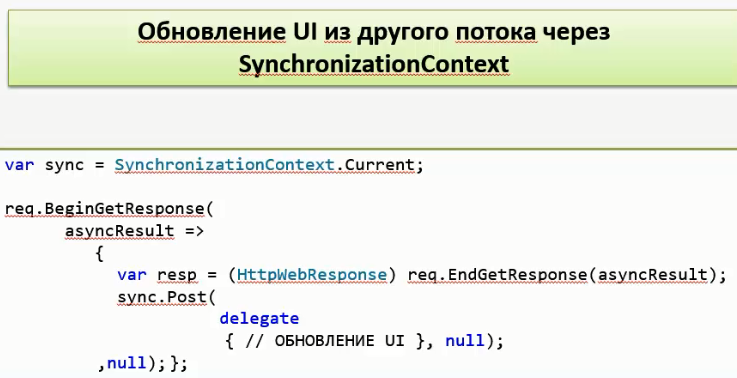


Асинхронно



SynchronizationContext

Для обращения к элементу управления через контекст синхронозации



var sync = SynchronizationContext.Current; - получения контекста синхронизации текущего потока

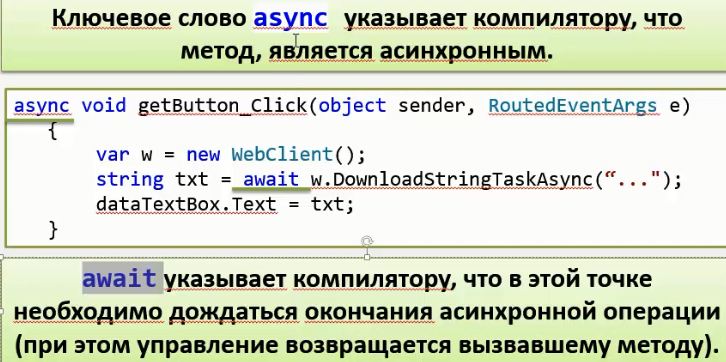
sync.Post( - метод для вызова асинхронного нотекста

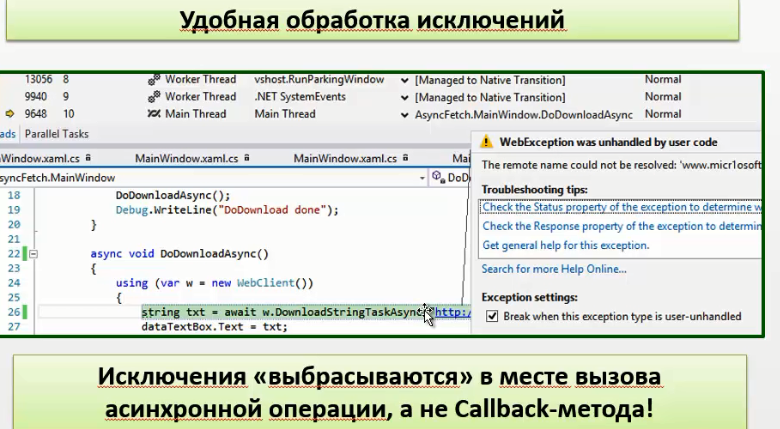
state => dataTextBox.Text += headersText,

null);

Новый асинхронный вызов методов

Суффикс методов должен быть TaskAsync или Async





var req = (HttpWebRequest) WebRequest.Create("http://www.microsoft.com");

req.Method = "GET";

var resp = (HttpWebResponse) await req.GetResponseAsync(); - горячая задача (запускается при ее создании)

пример



Asyn метод может вернуть только Task, Task<T> или void

СОБСТВЕННЫЙ AWAIT

async void CancelButtonClick(object sender, RoutedEventArgs e)

{

var t= new Task(() => Thread.Sleep(5000));

t.Start();

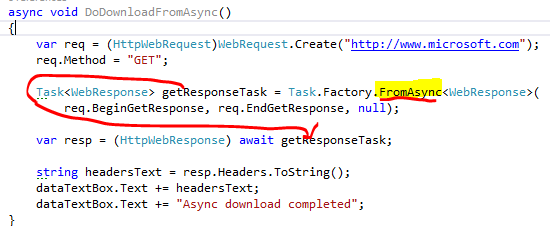
await t; или

await Task.Delay(2000);

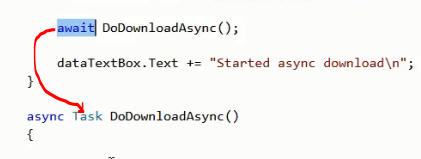
Для использования новой асинхронной модели в старых библиотеках Task.Factory.FromAsync

Task<WebResponse> getResponseTask = Task.Factory.FromAsync<WebResponse>(

req.BeginGetResponse, req.EndGetResponse, null); - ссылку на Begin End и дополнительные аргумент для CallBack метода

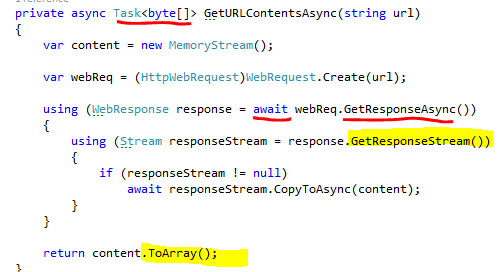


дожидаться можно только тех методов которые возвращают Task либо Task<T>



Возвратить результат



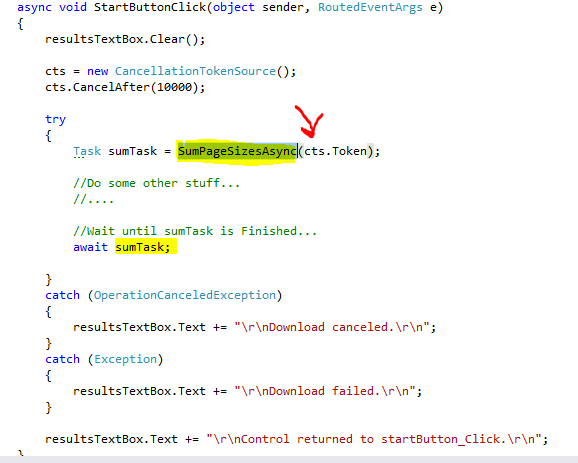


CancellationTokenSource

Отмена асинхронного метода во время его выполнения

cts.CancelAfter(10000); - автоматическая отмена через время

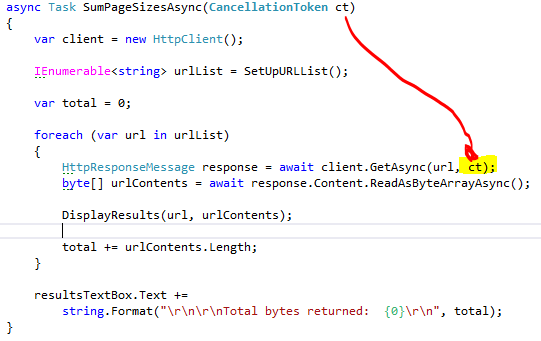
создаем отмену и передаем ее в метод



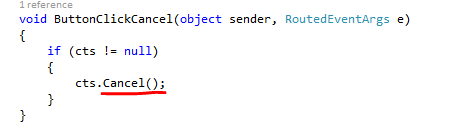
В теле метода

var client = new HttpClient();

HttpResponseMessage response = await client.GetAsync(url, ct); - перегруженный метод передаем отмену



И при нажатии на кнопку формы вызываем отмену



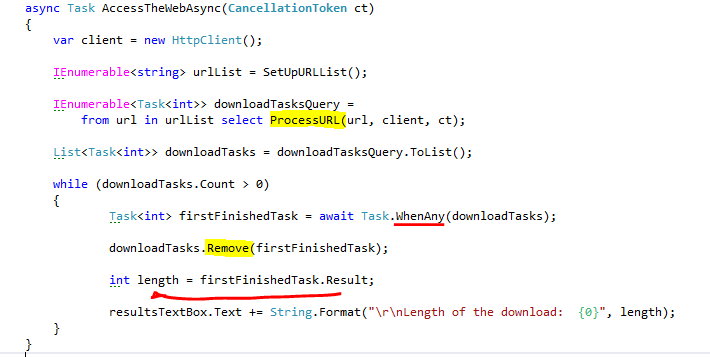
ТЕХНИКА ПАРАЛЛЕЛЬНОГО СЧИТЫВАНИЯ ИНФОРМАЦИИ С РЕСУРСА

В линк запросе запускаем метод ProcessURL(url, client, ct);

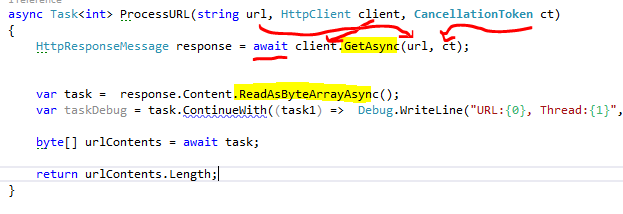
Task.WhenAny

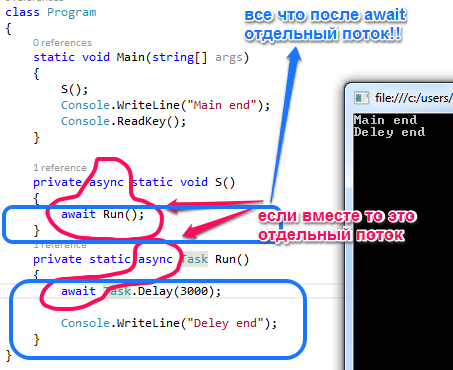
Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks); - дождаться любой первой задачи которая закончится и возвращает эту задачу

int length = await (await Task.WhenAny(task)); - сразу вернуть результат

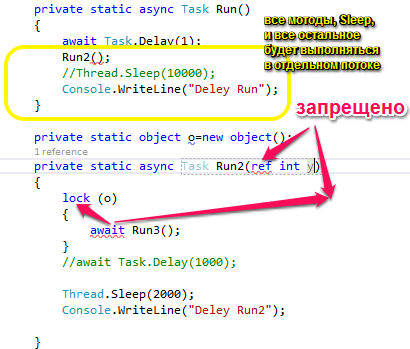


И в методе ждем пока считываются данные





Запрещено помещать await в блоки catch, finally



Task.Factory.FromAsync

Для методов использующих старую асинхронную модель Begin End

// Асинхронный прием клиентского подключения.

// ПРИМЕЧАНИЕ: NamedPipeServerStream использует старую модель

// асинхронного программирования.

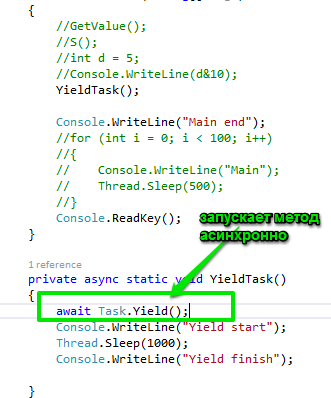
// Я преобразую ее к новой модели Task при помощи метода

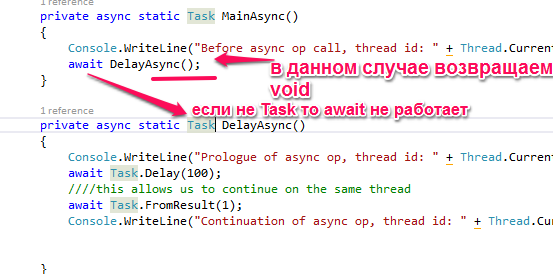
// FromAsync класса TaskFactory.

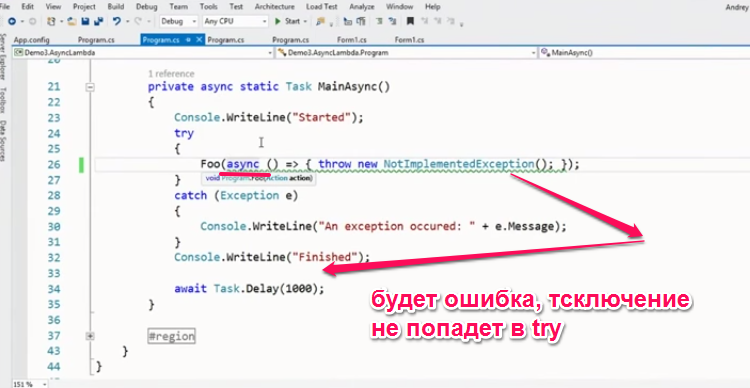
await Task.Factory.FromAsync(pipe.BeginWaitForConnection, [\\метод](file:///\\метод) старой модели

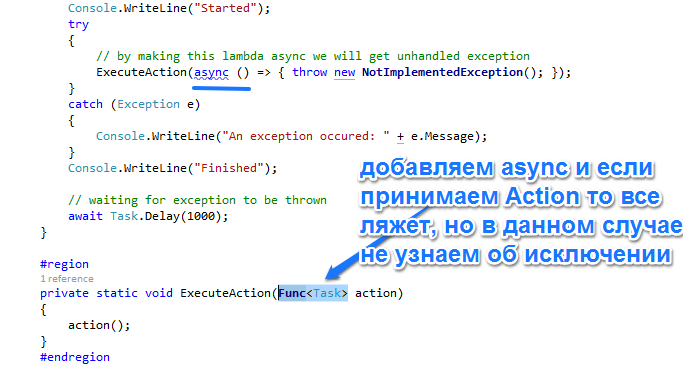
pipe.EndWaitForConnection, null); [\\конечный](file:///\\конечный) метод старой модели

Запуск метода асинхронно Task.Yield();

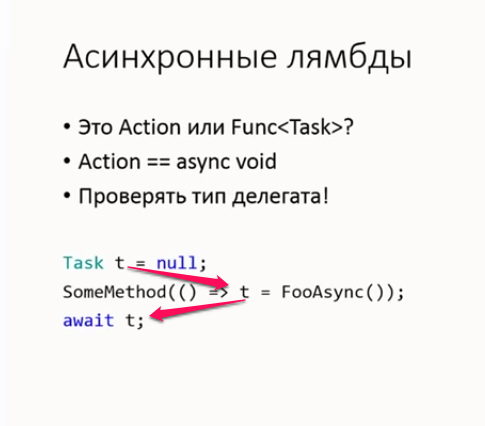




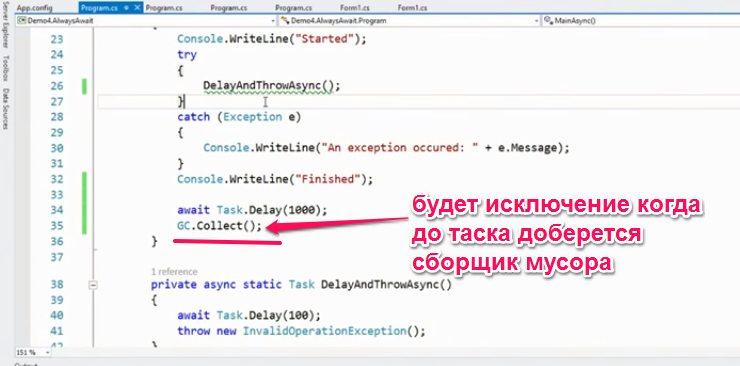




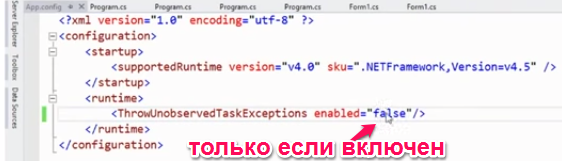
Нужно так



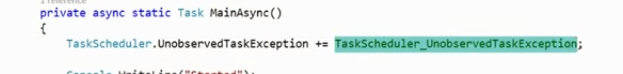
ТОЛЬКО 4.0 фреймверке



Или

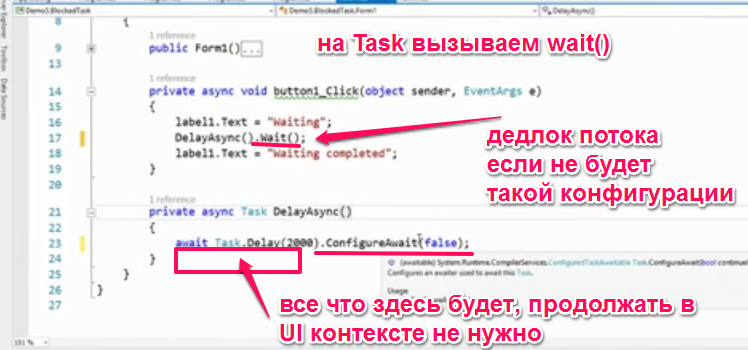


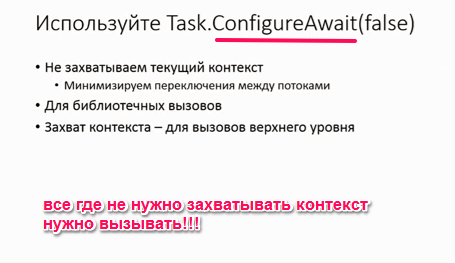
Либо подписаться на все исключения тасков

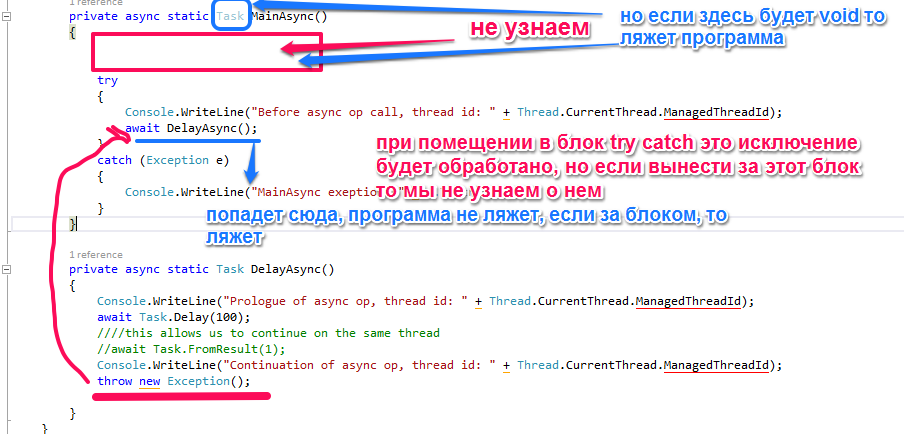


Говорит о том что текущий контекст захватывать не нужно, имеется в виду

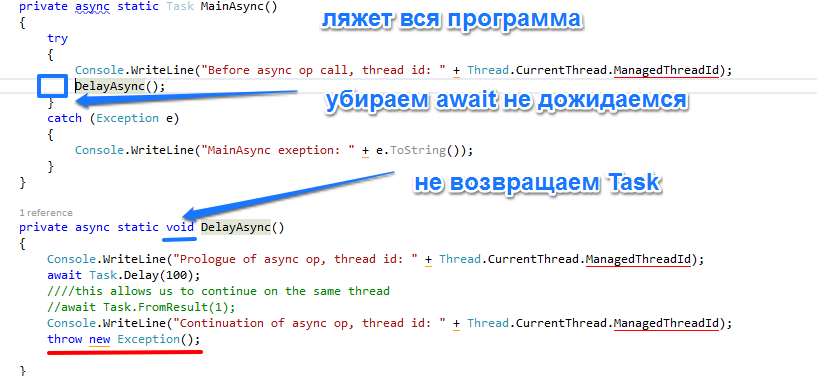
SynchronizationContext



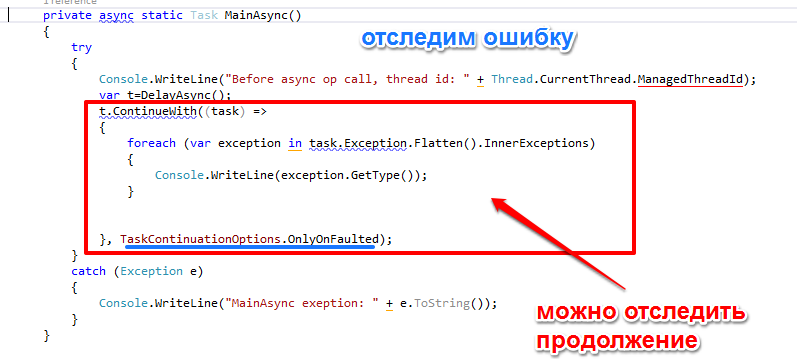




Если заменить void на Task то об исключении не узнаем

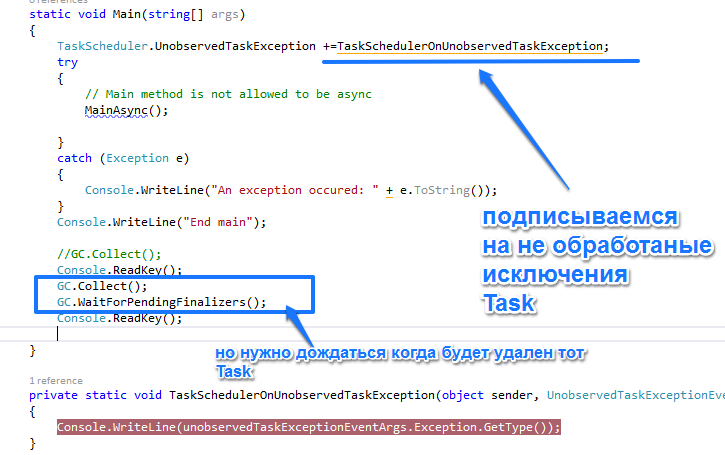


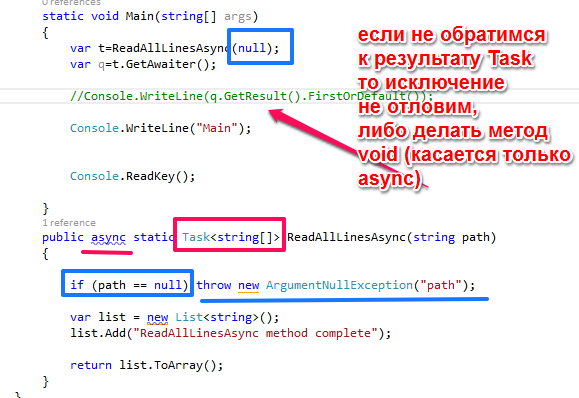
ОТСЛЕДИТЬ ОШИБКУ



TaskScheduler.UnobservedTaskException +=TaskSchedulerOnUnobservedTaskException;

Только если не обработанный Task





Взаимоблокировка

public static class DeadlockDemo {

private static async Task DelayAsync() {

await Task.Delay(1000);

}

public static void Test() {

var delayTask = DelayAsync();

delayTask.Wait();

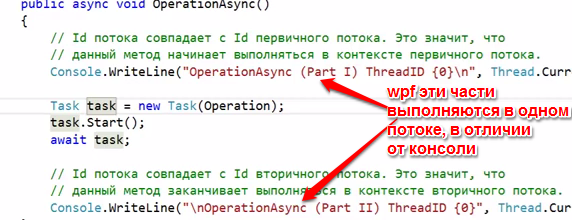
}

}

Данный код отлично работает в консольном приложении, но при вызове метода *DeadlockDemo.Test()*из GUI потока возникнет взаимоблокировка. Это связано с тем, как *await*обрабатывает контексты. По умолчанию, когда ожидается незавершенный *Task*, текущий контекст захватывается и используется для возобновления метода по окончании выполнения задачи. Контекстом является текущий *SynchronizationContext*, если только он не равен null, как в случае с консольными приложениями. Там это текущий *TaskScheduler*(контекст пула потоков).

GUI- и ASP.NET-приложения имеют*SynchronizationContext*, который разрешает единовременно выполнять только одну порцию кода. Когда выражение *await* завершает выполнение, оно пытается выполнить остальную часть *async*-метода в рамках захваченного контекста. Но он уже имеет поток, который (синхронно) ожидает завершения *async*-метода. Получается, что каждый из них ждет друг друга, вызывая взаимоблокировку.

Выполнение метода на разных платформах



Разбитие массива по частям

Console.WriteLine(string.Join("\r",q.GetResult()));

Работа HttpClient

