Multithreading

Thread

- Thread в винде:
  - Объект ядра потока (thread kernel object). контекст потока, набор регистров процессора ~ 1KB
  - Блок окружения потока (Thread Environment Block, TEB). 4KB, содержит заголовок цепочки обработки исключений, локальное хранилище данных для потока и некоторые структуры данных, используемые интерфейсом графических устройств (GDI) и графикой OpenGL.
  - Стек пользовательского режима (user-mode stack). По умолчанию на каждый стек Windows выделяет 1 Мбайт памяти.
  - Стек режима ядра (kernel-mode stack). x86 - 12 KB, X64 — 24 Кбайт.

System.Threading.Thread

- `Thread` - соответствует потоку в ОС
- Самый низкоуровневый объект в C# для работы с потоками
- Запрещен в приложениях для windows store

```
public static void Main()
{
    Console.WriteLine("Main thread");
    Thread dedicatedThread = new Thread(Compute);
    dedicatedThread.Start(5);
    Console.WriteLine("Main thread: Doing other work");
    Thread.Sleep(2000);     // Имитация другой работы
    dedicatedThread.Join(); // Ожидание завершения потока
    Console.WriteLine("Main thread: ending");
}

// Передаем делегат ParameterizedThreadStart в конструктор Thread
private static void Compute(Object state)
{
    Console.WriteLine("Compute: state={0}", state);
```

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

Методы:

- `Abort` уведомить CLR, что надо прекратить поток (для проверки завершенности следует опрашивать свойство `ThreadState`)
- `Interrupt` прервать поток на время
- `Join` остановить вызвающий поток до завершения потока, экземляру которого был вызван данный метод
- `Resume` возобновить работу потока
- `Start` запустить поток (при этом поток непосредственно создается в ОС)
- `Suspend` приостановить
- `static Sleep` останавливить поток
- `static GetDomain` ссылка на домен приложения
- `static GetDomainId` id текущего домена приложения

Приоритет потоков:

```
dedicatedThread.Priority = ThreadPriority.AboveNormal;
```

- Табличка, как связаны C# приоритеты потоков, приоритеты процессов в винде с реальными приоритетами потоков в ОС
- Левая колонка - приоритеты потоков в c#, Заголовки колонок - Process priority
- 17+ - драйвера устройств

CLR Priority	Idle	Below Normal	Normal	Above Normal	High	Realtime
Time-Critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above Normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below Normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22

CLR Priority	Idle	Below Normal	Normal	Above Normal	High	Realtime
Idle	1	1	1	1	1	16

В CLR все потоки делятся на foreground / background

- При завершении активного потока:
  - Принудительно завершит все фоновые потоки
  - Все фоновые потоки завершатся немедленно и без появления исключений
- `Thread` - по-умолчанию `foreground`, `ThreadPool` - background
- `IsBackground` - можно изменять в процессе работы
- Общая рекомендация - лучше использовать фоновые потоки

```
public static void Main()
{
    Thread t = new Thread(Worker);
    t.IsBackground = true;
    t.Start();

    // Активный поток (IsBackground = false) - приложение будет работать около 10
секунд
    // Фоновый поток (IsBackground = true) - немедленно прекратит работу
    // В LINQPad5 работает криво, в студии работает нормально :)
    Console.WriteLine("Returning from Main");
}
private static void Worker()
{
    Thread.Sleep(10000);
    Console.WriteLine("Returning from Worker");
}
```

Thread Pool

- Создавать потоки руками - очень низкоуровневый подход
- CLR умеет управлять собственным пулом потоков, чтобы не плодить лишние потоки
- На каждый объект CLR создается свой пул потоков, который используют все AppDomain
- Пулл потоков динамически определяет количество реальных потоков, которые необходимы приложению и сам добавляет / удаляет потоки в зависимости от того, какие задачи ставит приложение

```
public static class ThreadPool
{
    static Boolean QueueUserWorkItem(WaitCallback callBack);
    static Boolean QueueUserWorkItem(WaitCallback callBack, Object state);

    delegate void WaitCallback(Object state);
    ...
}
```

Базовый пример:

```
public static void Main()
{
    Console.WriteLine("Main thread: starting");
    ThreadPool.QueueUserWorkItem(Compute, 5);

    Console.WriteLine("Main thread: 10 sec waiting");
    Thread.Sleep(10000);

    Console.WriteLine("Main thread: exit");
}

private static void Compute(Object state)
{
    Console.WriteLine($"Compute: state = {state}");
    Thread.Sleep(1000);
}
```

Cancellation

- В C# используется стандартный паттерн отмены операций скоординированная отмена - оба класса должны явно поддерживать отмену
- `CancellationTokenSource` - специальный класс для управления токеном и непосредственно отменой операции

```
public sealed class CancellationTokenSource : IDisposable
{
    public CancellationTokenSource();
    public Boolean IsCancellationRequested { get; }

    // Токен, который передается в параллельный поток, считывая его свойства,
поток может реагировать на отмену и прекращать действие
    public CancellationToken Token { get; }

    // Отменить операцию!
    public void Cancel(); // Вызывает Cancel с аргументом false
    public void Cancel(Boolean throwOnFirstException);
    ...
}
```

- `CancellationToken` - класс, который передается в параллельный поток и по-которому можно понять, отменили задачу или еще нет.

```
public struct CancellationToken // Значимый тип
{
    // Статическое поле, используется, когда мы не хотим отменять метод, но он
поддерживает прием CancellationToken - передаем ему CancellationToken.None
    public static CancellationToken None { get; }

    // Непосредственные методы для реагирования на отмену
    Boolean IsCancellationRequested { get; }
    public void ThrowIfCancellationRequested();

    // Позволяет регистрировать дополнительные делегаты на событие отмены
    public CancellationTokenRegistration Register(Action<Object> callback, Object
state, Boolean useSynchronizationContext);

    // опущены другие методы, более простые перегрузки Register, GetHashCode,
Equals, == и != и прочее
}
```

Пример использования `CancellationTokenSource` и `CancellationToken`:

```
public static void Main()
{
    using (CancellationTokenSource cts = new CancellationTokenSource())
    {
        ThreadPool.QueueUserWorkItem(o => Count(cts.Token, 1000));
        Thread.Sleep(1000);
        cts.Cancel(); // Если метод Count уже вернул управления, Cancel не
оказывает никакого эффекта

        Thread.Sleep(1000);
        Console.WriteLine("Quit the programm");
    }
}

private static void Count(CancellationToken token, Int32 countTo)
{
    for (Int32 count = 0; count < countTo; count++)
    {
        if (token.IsCancellationRequested)
        {
```

```
            Console.WriteLine("Count is cancelled");
            break;
        }
        Console.WriteLine(count);
        Thread.Sleep(200);
    }
    Console.WriteLine("Count is done");
}
```

Пример регистрации дополнительных событий на отмену действия:

```
public static void Main()
{
    CancellationTokenSource cts = new CancellationTokenSource();
    CancellationToken token = cts.Token;

    var obj1 = new CancelableObject("1");
    var obj2 = new CancelableObject("2");

    token.Register(() => obj1.Cancel());
    token.Register(() => obj2.Cancel());

    cts.Cancel();
    cts.Dispose();
}

class CancelableObject
{
    public string id;

    public CancelableObject(string id)
```

```
    {
        this.id = id;
    }

    public void Cancel()
    {
        Console.WriteLine("Object {0} Cancel callback", id);
    }
}
```

Несколько `CancellationTokenSource` можно объединить в один и обрабатывать несколько причин отмены сразу:

```
var cts1 = new CancellationTokenSource();
cts1.Token.Register(() => Console.WriteLine("cts1 canceled"));
var cts2 = new CancellationTokenSource();
cts2.Token.Register(() => Console.WriteLine("cts2 canceled"));

var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(cts1.Token,
cts2.Token);
linkedCts.Token.Register(() => Console.WriteLine("linkedCts canceled"));

cts2.Cancel(); // Отмена любого из дочерних приводит к отмене общего

Console.WriteLine("cts1 canceled={0}, cts2 canceled={1}, linkedCts={2}",
    cts1.IsCancellationRequested, cts2.IsCancellationRequested,
linkedCts.IsCancellationRequested);
```

Есть стандартные методы, для отмены операций по Тайм-ауту:

```
public sealed class CancellationTokenSource : IDisposable
{
    public CancellationTokenSource(Int32 millisecondsDelay);
    public CancellationTokenSource(TimeSpan delay);
    public void CancelAfter(Int32 millisecondsDelay);
    public void CancelAfter(TimeSpan delay);
}
```

TPL

`System.Threading.Tasks`

Класс Task

- с ThreadPool есть глобальные проблемы:
  - возврат результатов из потока
  - как узнать о завершении операции
- Упрощенно: `Task` - типизированная обертка над пуллом потоков с кучей удобных методов
- Напоминаю, что все Task - берутся из пула и фоновые

```
ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);

new Task(ComputeBoundOp, 5).Start();

Task.Run(() => ComputeBoundOp(5));
```

Простейший пример создания задачи:

```
Task taskA = Task.Run( () => Console.WriteLine("Hello from thread '{0}'.",
Thread.CurrentThread.ManagedThreadId ));

Console.WriteLine("Hello from thread '{0}'.", Thread.CurrentThread.ManagedThreadId
);
taskA.Wait();
```

Есть типизированная версия `Task<TResult>` для возвращения конкретного результата:

```
// Создание задания Task (автоматически оно пока не выполняется)
Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000000);

t.Start();  // Можно начать выполнение задания через некоторое время

t.Wait();   // Можно ожидать завершения задания в явном виде
            // !!!!!! Существует перегруженная версия, принимающая тайм-
аут/CancellationToken
```

```
Console.WriteLine("The Sum is: " + t.Result); // Получение результата (свойство
Result вызывает метод Wait)
```

- `.Result`
  - Автоматически внутри вызывает метод `.Wait()`
- `.Wait()`
  - если метод еще не начал выполняться - может выполнять его прямо в текущем потоке, что потенциально может приводить к дедлокам
- Исключения, сделанные в методах задачи, сохраняются в отдельную коллекцию и при вызове `.Wait()` / `.Result` возвращаются исходному коду в виде `AggregateException`, который будет содержать коллекцию со всеми исключениями
  - Если не вызвать wait / result, то основной поток не узнает об ошибке

Пример, когда мы запускаем несколько задач и ждем, пока завершатся все:

```
Task[] tasks = new Task[3]
{
    new Task(() => Console.WriteLine("First")),
    new Task(() => Console.WriteLine("Second")),
    new Task(() => Console.WriteLine("Third"))
};
foreach (var t in tasks)
{
    t.Start();
}
Task.WaitAll(tasks);

Console.WriteLine("End");
```

Continuation

- `ContinueWith` - позволяет сразу настроить продолжение действия, вместо блокирования текущего потока
- возвращает `Task`, который частенько не используется

```
public static void Main()
{
    Task task1 = new Task(()=>{ Console.WriteLine($"current: {Task.CurrentId}");
});
    Task task2 = task1.ContinueWith(Display);
    Task task3 = task1.ContinueWith((Task t) => { Console.WriteLine($"current:
{Task.CurrentId}"); });
    Task task4 = task2.ContinueWith((Task t) => { Console.WriteLine($"current:
{Task.CurrentId}"); });

    task1.Start();
    task1.Wait(); // Будет ждать только task1, а не всю цепочку!
    Console.WriteLine("After task1 wait");
    Thread.Sleep(5000);
    Console.WriteLine("End");
```

```
}

static void Display(Task t)
{
    Console.WriteLine($"current: {Task.CurrentId}, previous: {t.Id} ");
    Thread.Sleep(3000);
}
```

Пример отмены задания с выбрасыванием исключения и обработкой такой ситуации в основном коде:

```
public static void Main()
{
    CancellationTokenSource cts = new CancellationTokenSource();
    Task<int> t = new Task<int>(() => Sum(cts.Token, 2), cts.Token);

    t.Start();

    cts.Cancel(); // кстати задача уже может быть завершена
    try
    {
        // В случае отмены задания именно метод Result генерирует исключение
AggregateException
        Console.WriteLine("The sum is: " + t.Result);
    }
    catch (AggregateException x)
    {
        // Пометить все OperationCanceledException ошибки, как обработанные,
        // остальные записать в новый объект AggregateException, выбросить его
заново, если он не пуст
        x.Handle(e => e is OperationCanceledException);
```

```
        Console.WriteLine("Sum was canceled");  // Строка выполняется, если все
исключения уже обработаны
    }
}
```

```
private static Int32 Sum(CancellationToken ct, int n)
{
    int sum = 0;
    for (; n > 0; n--)
    {
        ct.ThrowIfCancellationRequested(); // исключение
OperationCanceledException
        checked { sum += n; }
        // исключение System.OverflowException
    }
    return sum;
}
```

async / await

- Проблемы чистых Task
  - `.Result` блокирует поток, что не хорошо.
  - Писать реальный код с `ContinueWith` очень сложно и код получается тяжелый. Пример msdn

Если совсем упрощенно:

```
Factory.StartNew(() => DoSomeAsyncWork())
    .ContinueWith(
        (antecedent) =>
        {
            DoSomeWorkAfter();
        },
        TaskScheduler.FromCurrentSynchronizationContext());
```

Заменяется:

```
await DoSomeAsyncWork();
DoSomeWorkAfter();
```

- Ключевое слово `async`
  - Включает в методе поддержку await
  - Изменяет как обрабатывается результат
  - Не создает отдельных потоков или любой другой магии
- Ключевое слово `await`
  - Что-то в духе `asynchronous wait`
  - Некоторый унарный оператор, который принимает асинхронную операцию
  - Откладывает остаток метода до завершения операции, если она еще не выполнена
  - То есть метод ставится на паузу (что-то похожее на yielding) до завершения операции
  - Поток не блокируется

```
public async Task DoSomethingAsync()
{
  // In the Real World, we would actually do something...
  // For this example, we're just going to (asynchronously) wait 100ms.
  await Task.Delay(100);
}
```

```
async Task<int> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();

    // GetStringAsync returns a Task<string>. That means that when you await the
    // task you'll get a string (urlContents).
    Task<string> getStringTask =
client.GetStringAsync("http://msdn.microsoft.com");

    // You can do work here that doesn't rely on the string from GetStringAsync.
    DoIndependentWork();

    // The await operator suspends AccessTheWebAsync.
    //  - AccessTheWebAsync can't continue until getStringTask is complete.
    //  - Meanwhile, control returns to the caller of AccessTheWebAsync.
    //  - Control resumes here when getStringTask is complete.
    //  - The await operator then retrieves the string result from getStringTask.
    string urlContents = await getStringTask;

    return urlContents.Length;
}
```

```
public async Task NewStuffAsync()
{
  // Use await and have fun with the new stuff.
  await ...
}

public Task MyOldTaskParallelLibraryCode()
{
  // Note that this is not an async method, so we can't use await in here.
  ...
}

public async Task ComposeAsync()
{
  // We can await Tasks, regardless of where they come from.
  await NewStuffAsync();
  await MyOldTaskParallelLibraryCode();
}
```

- Что может возвращать `async` метод
  - `Task<T>`
  - `Task`
  - void

Всегда лучше возвращать Task вместо void, который используется в очень специфичных сценариях типа EventHandler / `static async void MainAsync()`

```
public async Task<int> CalculateAnswer()
{
  await Task.Delay(100); // (Probably should be longer...)

  return 42; // Return a type of "int", not "Task<int>"
}
```

Before	After	Comments
`task.Wait`	`await task`	Wait/await for a task to complete
`task.Result`	`await task`	Get the result of a completed task
`Task.WaitAny`	`await Task.WhenAny`	Wait/await for one of a collection of tasks to complete
`Task.WaitAll`	`await Task.WhenAll`	Wait/await for every one of a collection of tasks to complete
`Thread.Sleep`	`await Task.Delay`	Wait/await for a period of time
Task constructor	`Task.Run` or `TaskFactory.StartNew`	Create a code-based task

Context

```
// WinForms example (it works exactly the same for WPF).
private async void DownloadFileButton_Click(object sender, EventArgs e)
{
  // Since we asynchronously wait, the UI thread is not blocked by the file
download.
  await DownloadFileAsync(fileNameTextBox.Text);

  // Since we resume on the UI context, we can directly access UI elements.
  resultTextBox.Text = "File downloaded!";
}
```

```
private async Task DownloadFileAsync(string fileName)
{
  // Use HttpClient or whatever to download the file contents.
  var fileContents = await
DownloadFileContentsAsync(fileName).ConfigureAwait(false);

  // Note that because of the ConfigureAwait(false), we are not on the original
context here.
  // Instead, we're running on the thread pool.

  // Write the file contents out to a disk file.
  await WriteToDiskAsync(fileName, fileContents).ConfigureAwait(false);
  // The second call to ConfigureAwait(false) is not *required*, but it is Good
Practice.
}

// WinForms example (it works exactly the same for WPF).
private async void DownloadFileButton_Click(object sender, EventArgs e)
{
  // Since we asynchronously wait, the UI thread is not blocked by the file
download.
  await DownloadFileAsync(fileNameTextBox.Text);
```

```
    // Since we resume on the UI context, we can directly access UI elements.
    resultTextBox.Text = "File downloaded!";
}
```

- В .NET Core контекста нет
  - Вся информация, которая раньше хранилась в контексте, теперь передается через DI
- В .NET Framework контекст есть
  - Если нам не нужен контекс смело везде фигачим `.ConfigureAwait(false)`
- Если .NET Standard библиотека предполагает широкое использование (в том числе на .NET Framework), лучше `.ConfigureAwait(false)` везде, где это возможно.

```
public async Task DoOperationsConcurrentlyAsync()
{
  Task[] tasks = new Task[3];
  tasks[0] = DoOperation0Async();
  tasks[1] = DoOperation1Async();
  tasks[2] = DoOperation2Async();

  // At this point, all three tasks are running at the same time.

  // Now, we await them all.
  await Task.WhenAll(tasks);
}
```

```
public async Task<int> GetFirstToRespondAsync()
{
  // Call two web services; take the first response.
  Task<int>[] tasks = new[] { WebService1Async(), WebService2Async() };

  // Await for the first one to respond.
  Task<int> firstTask = await Task.WhenAny(tasks);

  // Return the result.
  return await firstTask;
}
```

```
await someObject;
```

```
private class FooAsyncStateMachine : IAsyncStateMachine
{
    // Member fields for preserving “locals” and other necessary state
    int $state;
    TaskAwaiter $awaiter;
    …
    public void MoveNext()
    {
        // Jump table to get back to the right statement upon resumption
        switch (this.$state)
        {
            …
            case 2: goto Label2;
            …
        }
        …
        // Expansion of “await someObject;”
        this.$awaiter = someObject.GetAwaiter();
```

```
        if (!this.$awaiter.IsCompleted)
        {
            this.$state = 2;
            this.$awaiter.OnCompleted(MoveNext);
            return;
            Label2:
        }
        this.$awaiter.GetResult();
        …
    }
}
```

- `await` нельзя:
  - в property getter/setter
  - inside lock/synclock
  - inside catch/finally
  - some other situations
- `await someTask;` vs `someTask.Wait;`
- `task.Result` vs `task.GetAwaiter().GetResult()`
  - сами значения в позитивном сценарии абсолютно идентичны
  - при ошибке `.Result` - AggregationException, `GetResult` - конкретную ошибку

Links:

- async
  - C# Asynchronous programming
  - MSDN: Асинхронное программирование с использованием ключевых слов Async и Await (C#)
  - Async and await (Stephen Cleary)
  - Async/Await - Best Practices in Asynchronous Programming (Stephen Cleary)
  - Async/Await FAQ
  - MSDN: Asynchronous Programming - Async Performance: Understanding the Costs of Async and Await (By Stephen Toub | October 2011)
  - Teplyakov: Dissecting async
  - Jon skeet: тонна мыслей и примеров про async, которые тяжело осмыслить
- habr
  - Habrahabr: async C#
  - Habrahabr: Использование async и await в C# — лучшие практики
- Joseph Albahari (Teplyakov translate)
  - 3 - Thread, Cancellation, Lazy
  - 4 - Barrier, Locks
  - 5.1 PLINQ

    - 5.2 Parallel, TPL, Task, Параллельные коллекции, SpinLock и SpinWait
- SOF
    - SOF: What is the difference between asynchronous programming and multithreading?
    - SOF:Asynchronous vs synchronous execution, what does it really mean?
    - difference-between-await-and-continuewith, good example continue with problem

ExecutionContext

Не лишним будет упомянуть, что есть контекст выполнения потока 1, 2:

- SecurityContext
- HostExecutionContext
- CallContext
- SynchronizationContext

MSDN:

> ExecutionContext is one of those things that the vast majority of developers never need to think about. It's kind of like air: it's important that it's there, but except at some crucial times (e.g. when something goes wrong with it), we don't think about it being there. ExecutionContext is actually just a container for other contexts. Some of these other contexts are ancillary, while some are vital to the execution model of .NET, but they all follow the same philosophy I described for ExecutionContext: if you have to know they're there, either you're doing something super advanced, or something's gone wrong.

MS Docs:

> An execution context is the managed equivalent of a COM apartment. Within an application domain, the entire execution context must be transferred whenever a thread is transferred. This situation occurs during transfers made by the Thread.Start method, most thread pool operations, and Windows Forms thread marshaling through the Windows message pump. It does not occur in unsafe thread pool operations (such as the UnsafeQueueUserWorkItem method), which do not transfer the compressed stack.

```
public sealed class ExecutionContext : IDisposable, ISerializable
{
    public static System.Threading.ExecutionContext Capture ();
    public static void Run (System.Threading.ExecutionContext executionContext,
System.Threading.ContextCallback callback, object state);

    [SecurityCritical] public static AsyncFlowControl SuppressFlow();
    public static void RestoreFlow();
    public static Boolean IsFlowSuppressed();
    // Не показаны редко применяемые методы
}
```

- ExecutionContext is captured with the static `Capture()` method
- it's restored during the invocation of a delegate via the static `Run()` method

MSDN:

> All of the methods in the .NET Framework that fork asynchronous work capture and restore ExecutionContext in a manner like this (that is, all except for those prefixed with the word “Unsafe,” which are unsafe because they explicitly do not flow ExecutionContext). For example, when you use Task.Run, the call to Run captures the ExecutionContext from the invoking thread, storing that ExecutionContext instance into the Task object. When the delegate provided to Task.Run is later invoked as part of that Task’s execution, it’s done so via ExecutionContext.Run using the stored context. This is true for Task.Run, for ThreadPool.QueueUserWorkItem, for Delegate.BeginInvoke, for Stream.BeginRead, for DispatcherSynchronizationContext.Post, and for any other async API you can think of.

SyncronizationContext

Контекст синхронизации MSDN, MSDN ExecContext vs SyncContext, 3

- SynchronizationContext is just an abstraction, one that represents a particular environment you want to do some work in.
- Предоставляет способ размещения единицы работы в очереди контекста
- `SynchronizationContext.Current` - получает контекст синхронизации, привязанный к текущему потоку (синглтон в рамках потока). Хранится в данных потока
- Хранение счетчика незавершенных асинхронных операций. Это позволяет использовать асинхронные ASP.NET-страницы и любой другой хост, где нужен счетчик такого рода. В большинстве случаев значение счетчика увеличивается на 1, когда захватывается текущий SynchronizationContext, и уменьшается на 1, когда захваченный SynchronizationContext используется для размещения уведомления о завершении в очереди контекста.
- Изначально сделаны для работы с UI потоком.
- Thread 1 / Thread 2 -> SyncContext -> GUI

Какие бывают реализации:

- `null` - это контекст по-умолчанию ThreadPool
- AspNetSynchronizationContext (System.Web.dll) / LegacyAspNetSynchronizationContext
    - Делегат, помещаемый в очередь полученного AspNetSynchronizationContext, восстанавливает идентификацию и культуру исходной страницы, а затем напрямую выполняет делегат. Этот делегат вызывается напрямую, даже если он ставится в очередь «асинхронно» вызовом Post.
    - С концептуальной точки зрения, контекст AspNetSynchronizationContext весьма сложен.
    - Если в одном приложении одновременно выполняется несколько операций, AspNetSynchronizationContext обеспечит их поочередное выполнение (по одной за раз). Они могут выполняться любым потоком, но этот поток получит идентификацию и культуру исходной страницы.
- DispatcherSyncronizationContext (WPF)
- WindowsFormsSynchronizationContext

```
public static void DoWork()
{
    var sc = SynchronizationContext.Current;
    ThreadPool.QueueUserWorkItem(delegate
    {
        … // do work on ThreadPool
        sc.Post(delegate
        {
            … // do work on the original context
        }, null);
   });
}
```

> When you await a task, by default the awaiter will capture the current SynchronizationContext, and if there was one, when the task completes it'll Post the supplied continuation delegate back to that context, rather than running the delegate on whatever thread the task completed or rather than scheduling it to run on the ThreadPool. If a developer doesn't want this marshaling behavior, it can be controlled by changing the awaitable/awaiter that's used. Whereas this behavior is always employed when you await a Task or Task, you can instead await the result of calling task.ConfigureAwait(...).

```
// The important aspects of the SynchronizationContext APIclass
SynchronizationContext
{
  // Dispatch work to the context.
  void Post(..); // (asynchronously)
  void Send(..); // (synchronously)
  // Keep track of the number of asynchronous operations.
  void OperationStarted();
  void OperationCompleted();
  // Each thread has a current context.
  // If "Current" is null, then the thread's current context is
  // "new SynchronizationContext()", by convention.
  static SynchronizationContext Current { get; }
  static void SetSynchronizationContext(SynchronizationContext);
}
```

	Выполнение делегатов в определенном потоке	**Делегаты выполняются по одному за раз**	**Делегаты выполняются в порядке очереди**	**Send может напрямую вызывать делегат**	**Post может напрямую вызывать делегат**
Windows Forms	Да	Да	Да	Если вызывается из UI-потока	Никогда
WPF/Silverlight	Да	Да	Да	Если вызывается из UI-потока	Никогда
По умолчанию	Нет	Нет	Нет	Всегда	Никогда
ASP.NET	Нет	Да	Нет	Всегда	Всегда

Deadlock на SynchronizationContext

Stephen Cleary: DontBlockAsyncCode:

```
public static async Task<JObject> GetJsonAsync(Uri uri) // My "library" method.
{
  using (var client = new HttpClient())
  {
    var jsonString = await client.GetStringAsync(uri);
    return JObject.Parse(jsonString);
  }
}
// My "top-level" method.
public class MyController : ApiController
{
  public string Get()
  {
    var jsonTask = GetJsonAsync(...);
    return jsonTask.Result.ToString(); // DEADLOCKED
  }
}
```

1. The top-level method calls GetJsonAsync (within the UI/ASP.NET context).
2. GetJsonAsync starts the REST request by calling HttpClient.GetStringAsync (still within the context).
3. GetStringAsync returns an uncompleted Task, indicating the REST request is not complete.
4. GetJsonAsync awaits the Task returned by GetStringAsync. The context is captured and will be used to continue running the GetJsonAsync method later. GetJsonAsync returns an uncompleted Task, indicating that the GetJsonAsync method is not complete.
5. The top-level method synchronously blocks on the Task returned by GetJsonAsync. This blocks the context thread.
6. ... Eventually, the REST request will complete. This completes the Task that was returned by GetStringAsync.
7. The continuation for GetJsonAsync is now ready to run, and it waits for the context to be available so it can execute in the context.
8. Deadlock. The top-level method is blocking the context thread, waiting for GetJsonAsync to complete, and GetJsonAsync is waiting for the context to be free so it can complete.

```
public static async Task<JObject> GetJsonAsync(Uri uri)
{
  using (var client = new HttpClient())
  {
    var jsonString = await client.GetStringAsync(uri).ConfigureAwait(false);
    return JObject.Parse(jsonString);
  }
}
public class MyController : ApiController
{
  public async Task<string> Get()
  {
    var json = await GetJsonAsync(...);
    return json.ToString();
  }
}
```

Synchronization primitives

MS Docs Synchronization primitives:

- Monitor - grants mutually exclusive access to a shared resource by acquiring or releasing a lock on the object that identifies the resource.
- Mutex (.NET Framework and .NET Core) - like Monitor, grants exclusive access to a shared resource., Unlike Monitor, the Mutex class can be used for inter-process synchronization.
- SpinLock / SpinWait
- Semaphore, SemaphoreSlim - limit the number of threads that can access a shared resource or a pool of resources concurrently
- EventWaitHandle, AutoResetEvent, ManualResetEvent, ManualResetEventSlim
- Barrier

lock

lock - базовый keywork языка, как if / else

- The lock statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released.

```
lock (x) // where x is an expression of a reference type.
{
    // Your code...
}
```

- You can't use the await keyword in the body of a lock statement.

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

volatile

volatile modifier:

> The volatile keyword indicates that a field might be modified by multiple threads that are executing at the same time. The compiler, the runtime system, and even hardware may rearrange reads and writes to memory locations for performance reasons. Fields that are declared volatile are not subject to these optimizations. Adding the volatile modifier ensures that all threads will observe volatile writes performed by any other thread in the order in which they were performed. There is no guarantee of a single total ordering of volatile writes as seen from all threads of execution.

Stackoverflow lock vs volatile vs interlocked:

> volatile just ensures the two CPUs see the same data at the same time

Interlocked

Interlocked Class - Provides atomic operations for variables that are shared by multiple threads.

```
public static int Add (ref int location1, int value);
public static int Increment (ref int location);
public static int Decrement (ref int location);
public static int Exchange (ref int location1, int value);
```

Lazy<T>

```
class Foo
{
  public readonly Expensive Expensive = new Expensive();
  ...
}
class Expensive {  /* Предположим, что создание этого объекта является дорогим */
}
```

```
class Foo
{
  Expensive _expensive;
  public Expensive Expensive // Создать экземпляр класса Expensive отложенно
  {
    get
    {
      if (_expensive == null)
          _expensive = new Expensive();
      return _expensive;
    }
  }
  ...
}
```

С многопоточностью плохо

```
Expensive _expensive;
readonly object _expenseLock = new object();

public Expensive Expensive
{
  get
  {
    lock (_expenseLock)
    {
      if (_expensive == null)
          _expensive = new Expensive();
      return _expensive;
    }
  }
}
```

Как-то фигово добавили многопоточности

```
volatile Expensive _expensive;
public Expensive Expensive
{
  get
  {
    if (_expensive == null)
    {
      var expensive = new Expensive();
      lock (_expenseLock)
      {
        if (_expensive == null)
            _expensive = expensive;
      }
    }
    return _expensive;
  }
}
```

Самому этим заниматься, конечно не надо.

Lazy<T>

```
Lazy<Expensive> _expensive = new Lazy<Expensive>(() => new Expensive(), true);

public Expensive Expensive { get { return _expensive.Value; } }
```

Concurrent Collections

MS Docs

- `ConcurrentBag<T>` Thread-safe implementation of an unordered collection of elements.
- `ConcurrentDictionary<TKey,TValue>` Thread-safe implementation of a dictionary of key-value pairs.
- `ConcurrentQueue<T>` Thread-safe implementation of a FIFO (first-in, first-out) queue.
- `ConcurrentStack<T>` Thread-safe implementation of a LIFO (last-in, first-out) stack.
- `BlockingCollection<T>` Provides bounding and blocking functionality for any type that implements IProducerConsumerCollection. For more information, see BlockingCollection Overview.
- `IProducerConsumerCollection<T>` The interface that a type must implement to be used in a BlockingCollection.