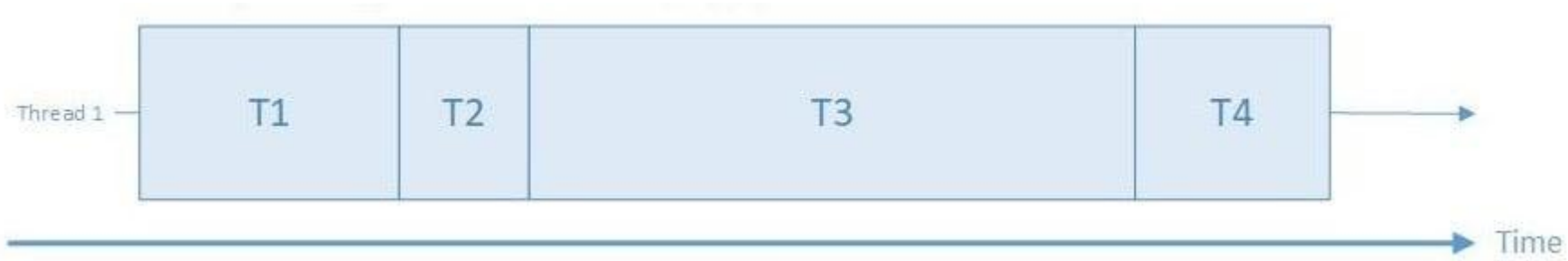
Многопоточность, Асинхроность

Однопоточность

Система в одном потоке работает со всеми задачами, выполняя их поочерёдно.

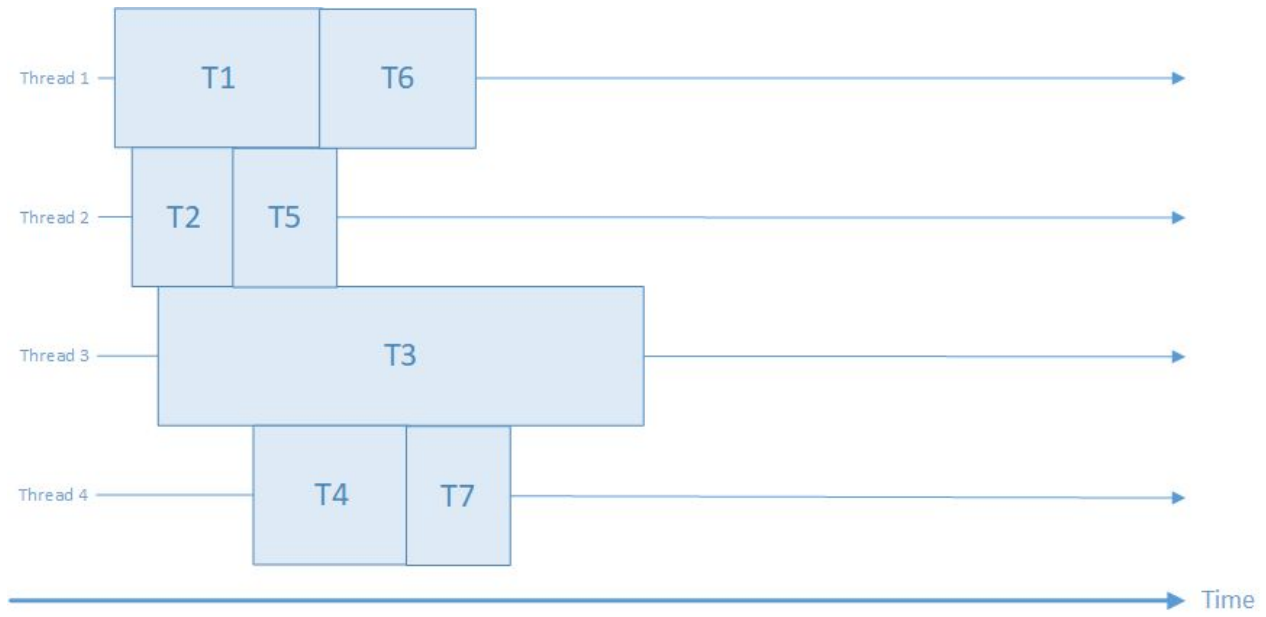


Многопоточность

В этом случае речь о нескольких потоках, в которых выполнение задач идет одновременно и независимо друг от друга.

Пример такого концепта — одновременная разработка веб- и мобильного приложений и серверной части, при условии соблюдения архитектурных «контрактов».

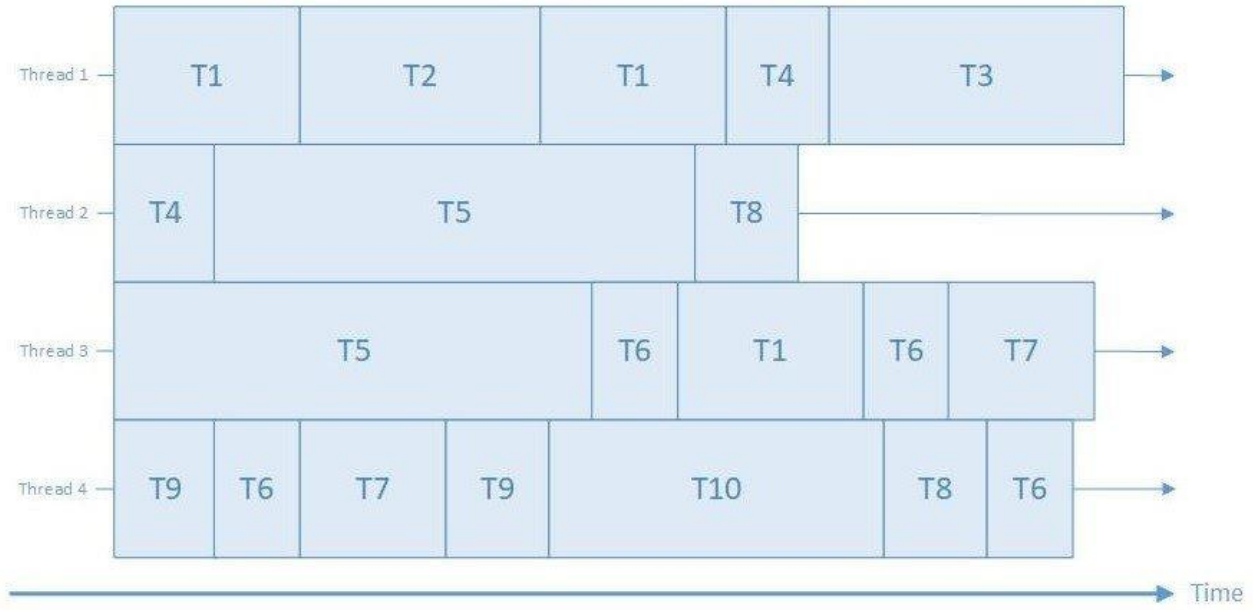
Использование нескольких потоков выполнения — один из способов обеспечить возможность реагирования приложения на действия пользователя при одновременном использовании процессора для выполнения задач между появлением или даже во время появления событий пользователя.



Асинхронность

Характеристики асинхронного кода:

- обрабатывает больше запросов сервера, предоставляя потокам возможность обрабатывать больше запросов во время ожидания результата от запросов ввода-вывода;
- делает пользовательский интерфейс быстрым, выделяя потоки для обработки действий в пользовательском интерфейсе во время ожидания запросов ввода-вывода, передавая затратные по времени операции другим ядрам ЦП.



Многопоточность VS Асинхронность

Многопоточность — параллельное выполнение, асинхронность — логическая оптимизация выполнения, которая может работать и в одном, и во многих потоках.

Проблемы многопоточности

Многозадачность

- Вытесняющая
- Кооперативная

Проблемы планирования задач

- Переключение контекста
- Приоритеты

Общая память

- Условия гонок (race condition)
- Взаимная блокировка (deadlock)

Асинхронный код в .NET

В .NET-фреймворке исторически сложилось несколько более старых паттернов организации асинхронного кода:

- APM (IAsyncResult, они же коллбеки) (.NET 1.0).
- EAP — события, этот паттерн все видели в WinForms (.NET 2.0).
- TAP (Task Asynchronous Pattern) — класс Task и его экосистема (.NET 4.0).

Класс System.Threading.Thread

Класс **Thread** является самым элементарным из всех типов в пространстве имен **System.Threading**. Он представляет объектно-ориентированную оболочку вокруг заданного пути выполнения внутри отдельного домена приложения. В этом классе определено несколько методов (статических и уровня экземпляра), которые позволяют создавать новые потоки внутри текущего домена приложения, а также приостанавливать, останавливать и уничтожать указанный поток.

Статические члены Thread

Статический член	Назначение
CurrentContext	Это свойство только для чтения возвращает контекст, в котором в текущий момент выполняется поток
CurrentThread GetDomain()	Это свойство только для чтения возвращает ссылку на текущий выполняемый поток
GetDomainID()	Эти методы возвращают ссылку на текущий домен приложения либо идентификатор домена, в котором выполняется текущий поток
Sleep ()	Этот метод приостанавливает текущий поток на указанное время

Члены уровня экземпляра Thread

Член уровня экземпляра	Назначение
IsAlive	Возвращает булевское значение, указывающее на то, запущен ли поток (и пока еще не прекращен или не отменен)
IsBackground	Получает или устанавливает значение, которое указывает, является ли данный поток фоновым (что более подробно объясняется далее в главе)
Name	Позволяет установить дружественное текстовое имя потока
Priority	Получает или устанавливает приоритет потока, который может принимать значение из перечисления ThreadPriority
ThreadState	Получает состояние данного потока, которое может принимать значение из перечисления ThreadState
Abort ()	Указывает среде CLR на необходимость как можно более скорого прекращения работы потока
Interrupt()	Прерывает (например, приостанавливает) текущий поток на подходящий период ожидания
Join ()	Блокирует вызывающий поток до тех пор, пока указанный поток (тот, на котором вызван метод Join ()) не завершится
Resume()	Возобновляет выполнение ранее приостановленного потока
Start()	Указывает среде CLR на необходимость как можно более скорого запуска потока
Suspend()	Приостанавливает поток. Если поток уже приостановлен, то вызов

Thread. Свойство Priority

- Lowest
- BelowNormal
- Normal
- AboveNormal
- Highest

Thread. Конструкторы

Thread(ParameterizedThreadStart) - Инициализирует новый экземпляр класса Thread, при этом указывается делегат, позволяющий объекту быть переданным в поток при запуске потока.

Thread(ThreadStart) - Инициализация нового экземпляра класса Thread.

Thread(ParameterizedThreadStart, Int32) - Инициализирует новый экземпляр класса Thread, при этом указывается делегат, позволяющий объекту быть переданным в поток при запуске потока с указанием максимального размера стека для потока.

Thread(ThreadStart, Int32) - Инициализирует новый экземпляр класса Thread, указывая максимальный размер стека для потока.

“ Talk is cheap. Show me the code. ”

Linus Torvalds

Thread. Запуск потока

```
0 references
public static void Main()
{
    for(int i = 0; i < 10; i++)
    {
        var thread = new Thread(() =>
        {
            var currentThread = Thread.CurrentThread;
            while (true)
            {
                Console.WriteLine($"Hello! My Id is :{currentThread.ManagedThreadId}");
                Thread.Sleep(5000);
            }
        });
        Thread.Sleep(1234);
        thread.Start();
    }
```

Thread. Запуск потока с параметрами

```
0 references
static void Main(string[] args)
{
    for (int i = 0; i < 10; i++)
    {
        var user = new User { Id = i + 1 };
        Thread thread = new Thread((object o) =>
        {
            var user = o as User;
            Thread currentThread = Thread.CurrentThread;
            while (true)
            {
                Console.WriteLine($"Hello. I am Thread and my id is {currentThread.ManagedThreadId}. My UserId is {user.Id}");
                Thread.Sleep(5000);
            }
        });

        thread.Start(user);
    }
}
```

Оператор lock

Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока.

Избегайте использования следующих объектов в качестве объектов блокировки:

- **this**, так как он может использоваться вызывающими объектами как блокировка;
- экземпляров Type, так как их может получать оператор typeof или отражение;
- строковых экземпляров, включая строковые литералы, так как они могут быть интернированы.

```
var  syncRoot = new object();

lock (syncRoot)
{
    //SomeCodeHere
}
```

C#

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

Monitor

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом **System.Threading.Monitor**. Фактически конструкция оператора **lock** инкапсулирует в себе синтаксис использования мониторов.

```
public static void Count()
{
    bool acquiredLock = false;
    try
    {
        Monitor.Enter(locker, ref acquiredLock);
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
    }
    finally
    {
        if(acquiredLock) Monitor.Exit(locker);
    }
}
```

Semaphore

Семафоры позволяют ограничить доступ определенным количеством объектов.

Его конструктор принимает два параметра: первый указывает, какому числу объектов изначально будет доступен семафор, а второй параметр указывает, какой максимальное число объектов будет использовать данный семафор.

Потоки вводят семафор, вызывая метод WaitOne(), который наследуется от класса WaitHandle, и освобождает семафор, вызывая метод Release().

Счетчик для семафора уменьшается каждый раз, когда поток входит в семафор, и увеличивается, когда поток освобождает семафор. Если значение счетчика равно нулю, последующие запросы блокируются до освобождения семафора другими потоками. Когда семафор освобожден всеми потоками, счетчик будет иметь максимальное значение, указанное при создании семафора.

Нет гарантированного порядка, например FIFO или LIFO, в котором заблокированные потоки вводят семафор.

Локальный семафор существует только в пределах процесса. Его может использовать любой поток в вашем процессе, имеющий ссылку на локальный объект Semaphore. Каждый объект Semaphore является отдельным локальным семафором.

Mutex

Mutex является классом-оболочкой над соответствующим объектом ОС Windows "мьютекс".

Когда двум или более потокам требуется одновременный доступ к общему ресурсу, системе необходим механизм синхронизации, гарантирующий, что ресурс будет использоваться только одним потоком в каждый момент времени. Mutex — это примитив синхронизации, предоставляющий эксклюзивный доступ к общему ресурсу только одному потоку. Если поток получает мьютекс, второй поток, желающий получить этот мьютекс, приостанавливается до тех пор, пока первый поток не освободит мьютекс.

Класс Mutex обеспечивает идентификацию потоков, поэтому мьютекс может быть освобожден только потоком, который его получил. В отличие от этого, класс Semaphore не применяет удостоверение потока. Мьютекс также может передаваться через границы домена приложения.

Основную работу по синхронизации выполняют методы **WaitOne()** и **ReleaseMutex()**.

Interlocked

Предоставляет атомарные операции для переменных, общедоступных нескольким потокам.

Член	Назначение
CompareExchange()	Безопасно проверяет два значения на равенство и, если они равны, то заменяет одно из значений третьим
Decrement()	Безопасно уменьшает значение на 1
Exchange()	Безопасно меняет два значения местами
Increment()	Безопасно увеличивает значение на 1

```
public void AddOne()
{
    int newVal = Interlocked.Increment(ref intVal);
}
```

ThreadPool

ThreadPool - Предоставляет пул потоков, который можно использовать для выполнения задач, отправки рабочих элементов, обработки асинхронного ввода-вывода, ожидания от имени других потоков и обработки таймеров.

Многие приложения создают потоки, которые тратят много времени на спящий режим, ожидая возникновения события. Другие потоки могут войти в спящее состояние только для периодического опроса на наличие изменений или сведений о состоянии обновления. Пул потоков позволяет более эффективно использовать потоки, предоставляя приложению пул рабочих потоков, управляемых системой

```
public class Example
{
    public static void Main()
    {
        // Queue the task.
        ThreadPool.QueueUserWorkItem(ThreadProc);
        Console.WriteLine("Main thread does some work, then sleeps.");
        Thread.Sleep(1000);

        Console.WriteLine("Main thread exits.");
    }

    // This thread procedure performs the task.
    static void ThreadProc(Object stateInfo)
    {
        // No state object was passed to QueueUserWorkItem, so stateInfo is null.
        Console.WriteLine("Hello from the thread pool.");
    }
}
// The example displays output like the following:
//       Main thread does some work, then sleeps.
//       Hello from the thread pool.
//       Main thread exits.
```

Параллельное программирование и библиотека TPL

В эпоху многоядерных машин, которые позволяют параллельно выполнять сразу несколько процессов, стандартных средств работы с потоками в .NET уже оказалось недостаточно. Поэтому во фреймворк .NET была добавлена библиотека параллельных задач TPL (Task Parallel Library), основной функционал которой располагается в пространстве имен **System.Threading.Tasks**. Данная библиотека позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах, если на целевом компьютере имеется несколько ядер. Кроме того, упрощается сама работа по созданию новых потоков. Поэтому начиная с .NET 4.0. рекомендуется использовать именно TPL и ее классы для создания многопоточных приложений, хотя стандартные средства и класс Thread по-прежнему находят широкое применение.

Task-based asynchronous pattern (TAP)

```
public async Task<int> MonthendProcessAsync(IList<int> Employees)
        1        2                          3
{
                    .....
  var Overtime = await new ERP().ProcessOvertime(emp);
                     4
                    .....
}
```

async/await

Ключевыми для работы с асинхронными вызовами в C# являются два ключевых слова: **async** и **await**, цель которых – упростить написание асинхронного кода. Они используются вместе для создания асинхронного метода.

Асинхонный метод обладает следующими признаками:

- В заголовке метода используется модификатор async
- Метод содержит одно или несколько выражений await
- В качестве возвращаемого типа используется один из следующих:
  - void
  - Task
  - Task<T>
  - ValueTask<T>

Как создать и запустить задачу?

1. **Фабрики запущенных задач**. Run — более легкая версия метода StartNew с установленными дополнительными параметрами по умолчанию. Возвращает созданную и запущенную задачу. Самый популярный способ запуска задач. Оба метода вызывают скрытый от нас Task.InternalStartNew. Возвращают объект Task.
2. **Фабрики завершенных задач**. Иногда нужно вернуть результат задачи без необходимости создавать асинхронную операцию. Это может пригодиться в случае подмены результата операции на заглушку при юнит-тестировании или при возврате заранее известного/рассчитанного результата.
3. **Конструктор**. Создает незапущенную задачу, которую вы можете далее запустить. Я не рекомендую использовать этот способ. Старайтесь использовать фабрики, если это возможно, чтобы не писать дополнительную логику по запуску.
4. **Фабрики-таскофикаторы**. Помогают либо произвести миграцию с других асинхронных моделей в TAP, либо обернуть логику ожидания результата в вашем классе в TAP. Например, FromAsync принимает методы паттерна APM в качестве аргументов и возвращает Task, который оборачивает более ранний паттерн в новый.

1. Фабрики запущенных задач	**3. Конструктор**
Task.Run(Action/Func) Task.Factory.StartNew(Action/Func)	var t = new Task(Action/Func); t.Start();
2. Фабрики завершенных задач	**4. Фабрики-таскофикаторы**
Task.FromResult(Result) Task.FromCanceled(CancellationToken) Task.FromException(Exception) Task.CompletedTask	Task.Factory.FromAsync (APM) TaskCompletionSource (EAP, APM, etc)

```
Task task = Task.Run(() => Console.WriteLine("Hello Task!"));
```

```
Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));
```

```
Task task = new Task(() => Console.WriteLine("Hello Task!"));
task.Start();
```

Отмена асинхронных операций

Для отмены асинхронных операций используются классы **CancellationToken** и **CancellationTokenSource**.

CancellationToken содержит информацию о том, надо ли отменять асинхронную задачу. Асинхронная задача, в которую передается объект **CancellationToken**, периодически проверяет состояние этого объекта. Если его свойство **IsCancellationRequested** равно **true**, то задача должна остановить все свои операции.

Для создания объекта **CancellationToken** применяется объект **CancellationTokenSource**. Кроме того, при вызове у CancellationTokenSource метода **Cancel()** у объекта **CancellationToken** свойство IsCancellationRequested будет установлено в true.

```
class Program
{
    static void Factorial(int n, CancellationToken token)
    {
        int result = 1;
        for (int i = 1; i <= n; i++)
        {
            if (token.IsCancellationRequested)
            {
                Console.WriteLine("Операция прервана токеном");
                return;
            }
            result *= i;
            Console.WriteLine($"Факториал числа {i} равен {result}");
            Thread.Sleep(1000);
        }
    }
    // определение асинхронного метода
    static async void FactorialAsync(int n, CancellationToken token)
    {
        if(token.IsCancellationRequested)
            return;
        await Task.Run(()=>Factorial(n, token));
    }

    static void Main(string[] args)
    {
        CancellationTokenSource cts = new CancellationTokenSource();
        CancellationToken token = cts.Token;
        FactorialAsync(6, token);
        Thread.Sleep(3000);
        cts.Cancel();
        Console.Read();
    }
}
```

Как следить за прогрессом выполнения?

TAP содержит специальный интерфейс для использования в своих асинхронных классах — **IProgress<T>**, где **T** — тип, содержащий информацию о прогрессе, например **int**. Согласно конвенциям, **IProgress** может передаваться как последние аргументы в метод вместе с **CancellationToken**. В случае если вы хотите передать только что-то из них, в паттерне существуют значения по умолчанию: для **IProgress** принято передавать **null**, а для **CancellationToken** — **CancellationToken.None**, так как это структура.

```
public async Task RunAsync(int delay, CancellationToken cancellationToken, IProgress<int> progress)
{
      int completePercent = 0;

      while (completePercent < 100)
      {
        await Task.Run(() =>
        {
            completePercent++;

             new Task(() =>
             {
                 progress?.Report(completePercent);
             }, cancellationToken,
                TaskCreationOptions.PreferFairness).Start();

        }, cancellationToken);

        await Task.Delay(delay, cancellationToken);
      }
}
```

Как синхронизировать задачи?

Комбинаторы задач Task.WaitAll (list of tasks) -> Task Task.WaitAny (list of tasks) -> Task Task.WhenAll (list of tasks) -> Task Task.WhenAny (list of tasks) -> Task	**Метод расширения ContinueWith для экземпляров задач с опциями реакции на исключения, отмену или удачное завершение предыдущей задачи** t.ContinueWith(res=>{ код продолжения }, TaskContinuationOptions)
Метод расширения ContinueWith для экземпляров задач с опциями продолжения синхронно или асинхронно, установкой другого планировщика задач t.ContinueWith(res=>{ код продолжения }, TaskContinuationOptions)	**Метод расширения ContinueWith для экземпляров задач с опциями присоединения к времени выполнения родительской задачи (дочерняя задача не сможет завершиться до завершения родительской)** t.ContinueWith(res=>{ код продолжения }, TaskContinuationOptions)

Task.WaitAll(tasks) и Task.WaitAny(tasks)

```
static void Main(string[] args)
{
    Task[] tasks1 = new Task[3]
    {
        new Task(() => Console.WriteLine("First Task")),
        new Task(() => Console.WriteLine("Second Task")),
        new Task(() => Console.WriteLine("Third Task"))
    };
    foreach (var t in tasks1)
        t.Start();
    Task.WaitAll(tasks1); // ожидаем завершения задач

    Console.WriteLine("Завершение метода Main");

    Console.ReadLine();
}
```

Как извлечь результат из задачи?

До появления **await** извлекать результат из задач можно было такими блокирующими способами:

- **t.Result();** — возврат результата / выброс исключения **AggregateException**.
- **t.Wait()**; — ожидание выполнения задачи, выброс исключения **AggregateException**.
- **t.GetAwaiter().GetResult()**; — возврат результата / выброс оригинального исключения — служебный метод компилятора, поэтому использовать его не рекомендуется. Используется механизмом **async/await**.

После появления **async/await** рекомендованной техникой стал оператор await, производящий неблокирующее ожидание. То есть если await добрался до незавершенной задачи, выполнение кода в потоке будет прервано и продолжится только с завершением задачи.

- **await t**; — возврат результата / выброс оригинального исключения.

Следует заметить, что для **t.GetAwaiter().GetResult()**; и await будет выброшено только первое исключение, аналогично манере поведения обычного синхронного кода.

Как запустить задачу и ожидать ее результат?

```
class Program
{
    static void Main(string[] args)
    {
        Task<int> task1 = new Task<int>(()=>Factorial(5));
        task1.Start();

        Console.WriteLine($"Факториал числа 5 равен {task1.Result}");

        Task<Book> task2 = new Task<Book>(() =>
        {
            return new Book { Title = "Война и мир", Author = "Л. Толстой" };
        });
        task2.Start();

        Book b = task2.Result;  // ожидаем получение результата
        Console.WriteLine($"Название книги: {b.Title}, автор: {b.Author}");

        Console.ReadLine();
    }


    static int Factorial(int x)
    {
        int result = 1;

        for (int i = 1; i <= x; i++)
        {
            result *= i;
        }

        return result;
    }
}

public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
}
```

Как запустить задачу и ожидать ее результат?

```
class Program
{
    static void Factorial(int n)
    {
        int result = 1;
        for (int i = 1; i <= n; i++)
        {
            result *= i;
        }
        Console.WriteLine($"Факториал равен {result}");
    }

    // определение асинхронного метода
    static async Task FactorialAsync(int n)
    {
        await Task.Run(()=>Factorial(n));
    }
    static void Main(string[] args)
    {
        FactorialAsync(5);
        FactorialAsync(6);
        Console.WriteLine("Некоторая работа");
        Console.Read();
    }
}
```

Исключения

```
class Program
{
    static void Factorial(int n)
    {
        if (n < 1)
            throw new Exception($"{n} : число не должно быть меньше 1");
        int result = 1;
        for (int i = 1; i <= n; i++)
        {
            result *= i;
        }
        Console.WriteLine($"Факториал числа {n} равен {result}");
    }

    static async void FactorialAsync(int n)
    {
        try
        {
            await Task.Run(() => Factorial(n));
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }

    static void Main(string[] args)
    {
        FactorialAsync(-4);
        FactorialAsync(6);

        Console.Read();
    }
}
```

```
static async Task DoMultipleAsync()
{
    Task allTasks = null;

    try
    {
        Task t1 = Task.Run(()=>Factorial(-3));
        Task t2 = Task.Run(() => Factorial(-5));
        Task t3 = Task.Run(() => Factorial(-10));

        allTasks = Task.WhenAll(t1, t2, t3);
        await allTasks;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Исключение: " + ex.Message);
        Console.WriteLine("IsFaulted: " + allTasks.IsFaulted);
        foreach (var inx in allTasks.Exception.InnerExceptions)
        {
            Console.WriteLine("Внутреннее исключение: " + inx.Message);
        }
    }
}
```

TaskCompletionSource

```
public static Task<bool> UploadFile(string name, byte[] data)
{
  var taskCompletionSource = new TaskCompletionSource<bool>();
  try
  {
    MyBox.UploadFile(name, data, success =>
    {
      taskCompletionSource.SetResult(success);
    });
  }
  catch (Exception ex)
  {
    taskCompletionSource.SetException(ex);
  }
  return taskCompletionSource.Task;
}
```

```
public async void OnUploadButtonClicked()
{
  statusText.Text = "Generating Image...";
  byte[] imageData = await GenerateImage();
  statusText.Text = "Uploading Image...";
  bool success = await MyBoxHelper.UploadFile("image.jpg", imageData);
  statusText.Text = success ? string.Empty : "Error Uploading";
}
```

Домашка

- Чем отличается SemaphoreSlim от Semaphore? Когда какой выбрать?
- Прочитать статью про **асинхронное программирование** (ссылка в «Что почитать»)
- Задание: С использованием **Thread**-ов написать приложение, реализующее патерн **Publisher/Subscriber**. Необходимо реализовать 2 класса **Publisher**, который генерирует случайные целые числа и складывает их в очередь. Класс **Subscriber**, который опрашивает очередь и вычитывает из нее числа и выводит их в консоль. Очередь должна быть потокобезопасная. В программе запуска может быть несколько экземпляров типа **Publisher**, которые одновременно генерируют случайные числа и складывают их в очередь. Должен быть только один объект типа **Subscriber** получающий данные из очереди.

Что почитать

- Параллелизм, многопоточность, асинхронность
- Многопоточное программирование и его проблемы
- TAP MSDN
- async/await
- Асинхронное программирование