Платформа параллельных вычислений

https://github.com/PatseiBSTU/PFX

Parallel Framework, PFX

- это набор типов и технологий, являющийся частью платформы .NET (Core), предназначен для повышения производительности за счёт средств, упрощающих добавление параллелизма в приложения
- обеспечивает три уровня организации параллелизма:
  - Параллелизм на уровне задач. Библиотека параллельных задач (Task Parallel Library, TPL).
    System.Threading.Tasks и System.Threading
  - Параллелизм при императивной обработке данных
  - Параллелизм при декларативной обработке данных реализуется при помощи параллельного интегрированного языка запросов (PLINQ).

- Ограничения класс Thread:

- 1) отсутствует механизм продолжений
- 2) затруднено получение значение результата из потока
- 3) повышенный расход памяти и замедление работы приложения

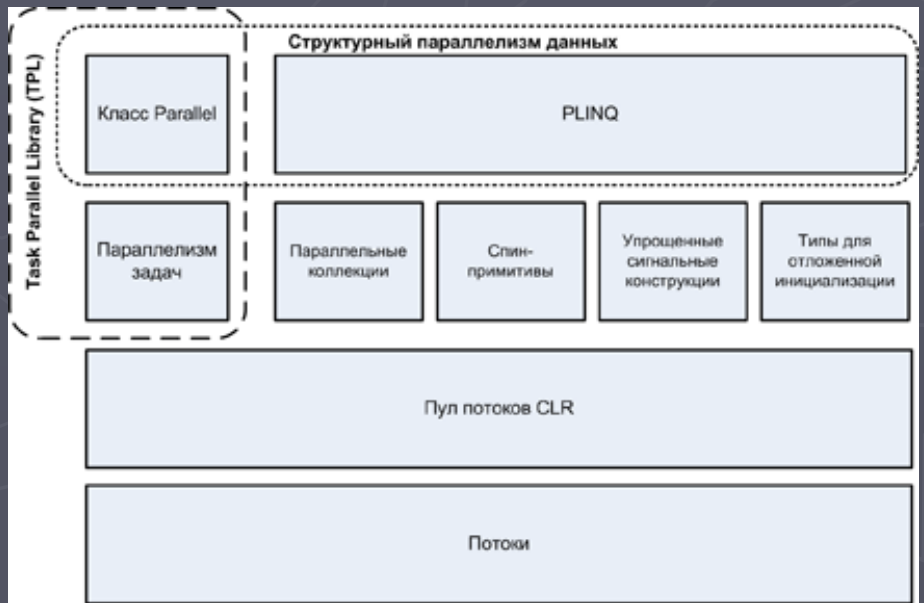
Библиотека параллельных задач TPL (Task Parallel Library)

позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах (для создания многопоточных приложений)

Задача (task) – абстракция более высокого уровня чем поток

using System.Threading.Tasks

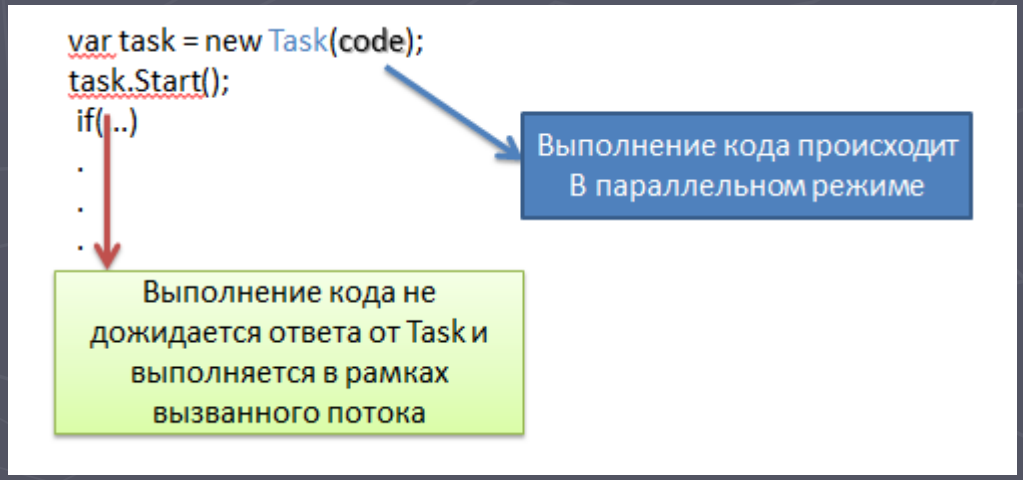
Планировщик библиотеки выполняет диспетчеризацию задач, а также предоставляет единообразный механизм отмены задач и обработки исключительных ситуаций



класс Task

- описывает отдельную продолжительную операцию, которая запускается асинхронно в одном из потоков из пула потоков (можно запускать синхронно в текущем потоке) – подобна потокам, но абстракция более высокого уровня

- Представлена .Net 4.0
- Среда WinRT



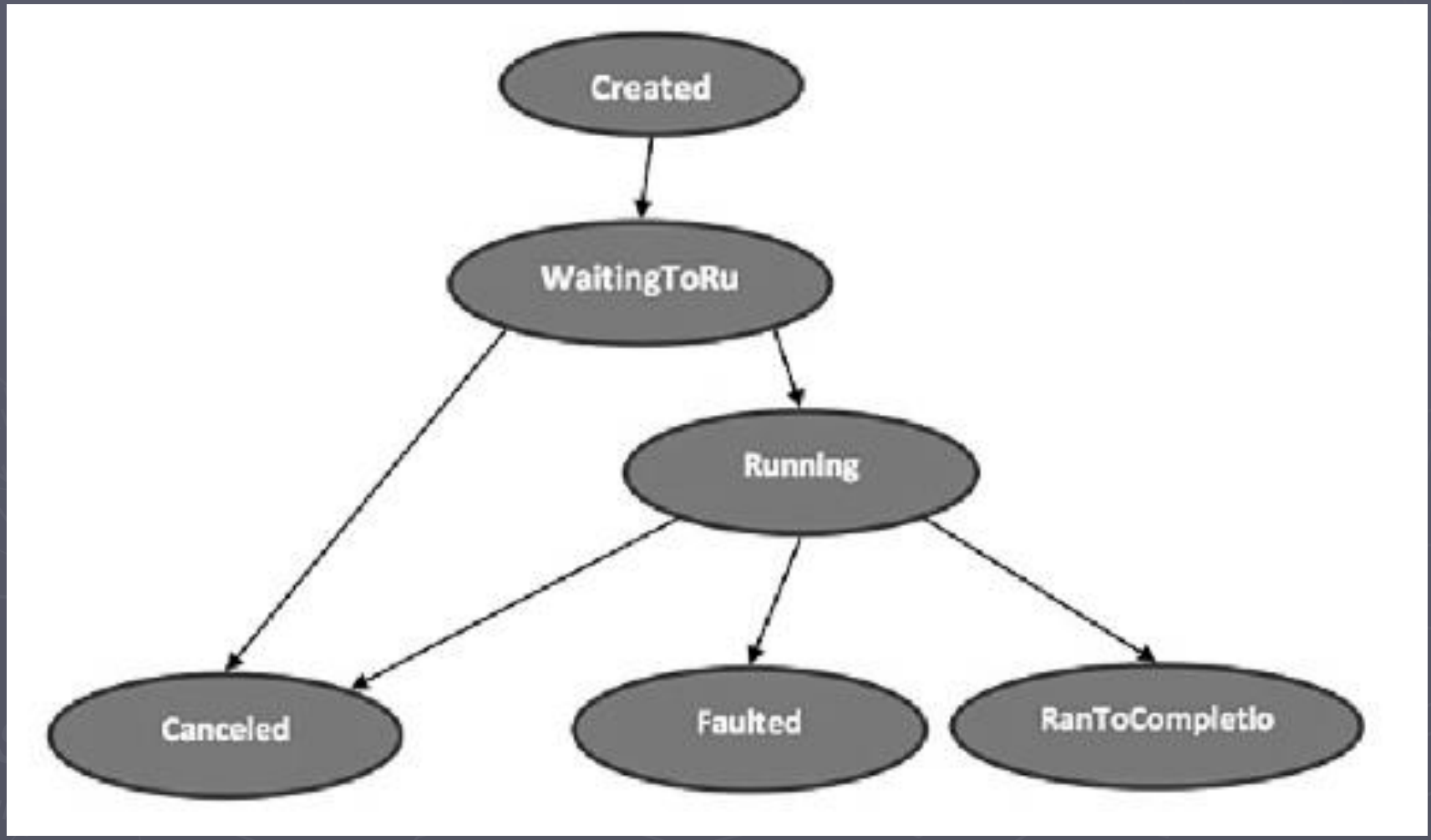
Элементы класса Task<T>

AsyncState	Объект, заданный при создании задачи (аргумент Action<object>)
ConfigureAwait()	Настраивает объект ожидания, используемый для текущей задачи
ContinueWith()	Используются для указания метода, выполняемого после завершения текущей задачи
CreationOptions	Опции, указанные при создании задачи (тип TaskCreationOptions)
CurrentId	Статическое свойство типа int?, которое возвращает целочисленный идентификатор текущей задачи
Delay()	Статический метод; позволяет создать задачу с указанной задержкой старта
Dispose()	Освобождение ресурсов, связанных с задачей
Exception	Возвращает объект типа AggregateException, который соответствует исключению, прервавшему выполнение задачи
Factory	Доступ к фабрике, содержащей методы создания Task и Task<T>
GetAwaiter()	Получает объект ожидания для текущей задачи
Id	Целочисленный идентификатор задачи

Элементы класса Task<T>

IsCanceled	Булево свойство, указывающее, была ли задача отменена
IsCompleted	Свойство равно true, если задача успешно завершилась
IsFaulted	Свойство равно true, если задача сгенерировала исключение
Run()	Статический метод; выполняет создание и запуск задачи
RunSynchronously()	Запуск задачи синхронно
Start()	Запуск задачи асинхронно
Status	Возвращает текущий статус задачи (объект типа TaskStatus)
Wait()	Приостанавливает текущий поток до завершения задачи
WaitAll()	Статический метод; приостанавливает текущий поток до завершения всех указанных задач
WaitAny()	Статический метод; приостанавливает текущий поток до завершения любой из указанных задач
WhenAll()	Статический метод; создаёт задачу, которая будет выполнена после выполнения всех указанных задач
WhenAny()	Статический метод; создаёт задачу, которая будет выполнена после выполнения любой из указанных задач

Состояния Task



Создание задачи

аргумент типа Action – метод, выполняемый в задаче

```
int i = 10;
Task task1 = new Task(() =>
                    { i++; Console.WriteLine("Task 1 finished"); });
task1.Start();
```

запускает задачу, вернее, помещает её в очередь запуска планировщика задач - асинхронный запуск

```
Task task2 = Task.Factory.StartNew(() =>
                    { ++i; Console.WriteLine("Task 2 finished"); });


Task task3 = Task.Run(() =>
                    { ++i; Console.WriteLine("Task 3 finished"); });
```

```
Console.WriteLine(i);
Console.WriteLine("Main finished");
```

```
10
Main finished
Task 2 finished
Task 3 finished
Task 1 finished
```

```
int i = 10;
Task task1 = new Task(() =>
                    { i++; Console.WriteLine("Task 1 finished"); });
task1.Start();


Task task2 = Task.Factory.StartNew(() =>
                    { ++i; Console.WriteLine("Task 2 finished"); });


Task task3 = Task.Run(() =>
                    { ++i; Console.WriteLine("Task 3 finished"); });
```

```
Task.WaitAll(task1, task2,task3);
```

```
Console.WriteLine(i);
Console.WriteLine("Main finished");
```

```
Task 2 finished
Task 1 finished
Task 3 finished
13
Main finished
```

Wait(), WaitAll() и WaitAny() останавливают основной поток до завершения задачи (или задач)
task1.Wait(1000);

- Задачи могут быть вложенные
- Запуск задачи

```
 int i = 10;
Task task3 = Task.Run(() =>
                    { ++i; Console.WriteLine("Task 3 finished"); });

// task3.Start(); // System.InvalidOperationException:
//Start нельзя вызывать для уже запущенной задачи.
```

- Синхронный запуск

```
 Action<object> method = x =>
                     { Thread.Sleep(1000);
                      Console.WriteLine(x.ToString()); };

var task4 = new Task(method, TaskCreationOptions.LongRunning);
task4.RunSynchronously();
```

задаёт вид задачи (например, LongRunning – долгая задача)

выполняет задачу синхронно

Возврат результата

- Task<TResult> - описывает задачу, возвращающую значение типа Tresult
- принимают аргументы типа
  - Func<TResult>
  - Func<object, TResult> (опционально – аргументы типа CancellationToken и TaskCreationOptions)

```
Func<int> func = () =>
                        { Thread.Sleep(1000);
                            return ++i; };
            Task<int> task = new Task<int>(func);
            Console.WriteLine(task.Status);          // Created
            task.Start();
            Console.WriteLine(task.Status);          // WaitingToRun
            task.Wait();
            Console.WriteLine(task.Result);          // 14
            Console.WriteLine("Main finished");
```

```
13
Created
WaitingToRun
14
Main finished
Для продолжения нажмите любую клавишу
```

Обработка исключений

- System.AggregateException

```
Task task5 = Task.Run(() => {
                                throw new Exception(); });
            try
            {
                task5.Wait();
            }
            catch (AggregateException ex)
            {
                var message = ex.InnerException.Message;
                Console.WriteLine(message);
            }
```

Отмена выполнения задач

- Структура CancellationToken - токен отмены

```
CancellationTokenSource tokenSource =
                    new CancellationTokenSource();
// используем  токен в двух задачах
new Task(method, tokenSource.Token).Start();
new Task(method, tokenSource.Token).Start();

//  отменяем  задачи
tokenSource.Cancel();
```

Продолжения (**continuation task**)

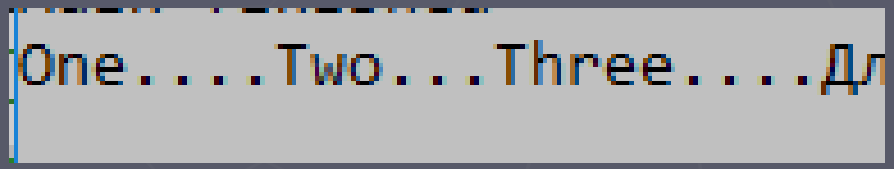
- сообщает задаче, что после её завершения она должна продолжить делать что-то другое

```
Task task6 = Task.Run(() =>
                    Console.Write("Doing.."));

Task task7 = task6.ContinueWith(t =>
                  Console.Write("continuation"));
```

После того как задача завершается, отказывает или отменяется, задача task7 (продолжение) запускается

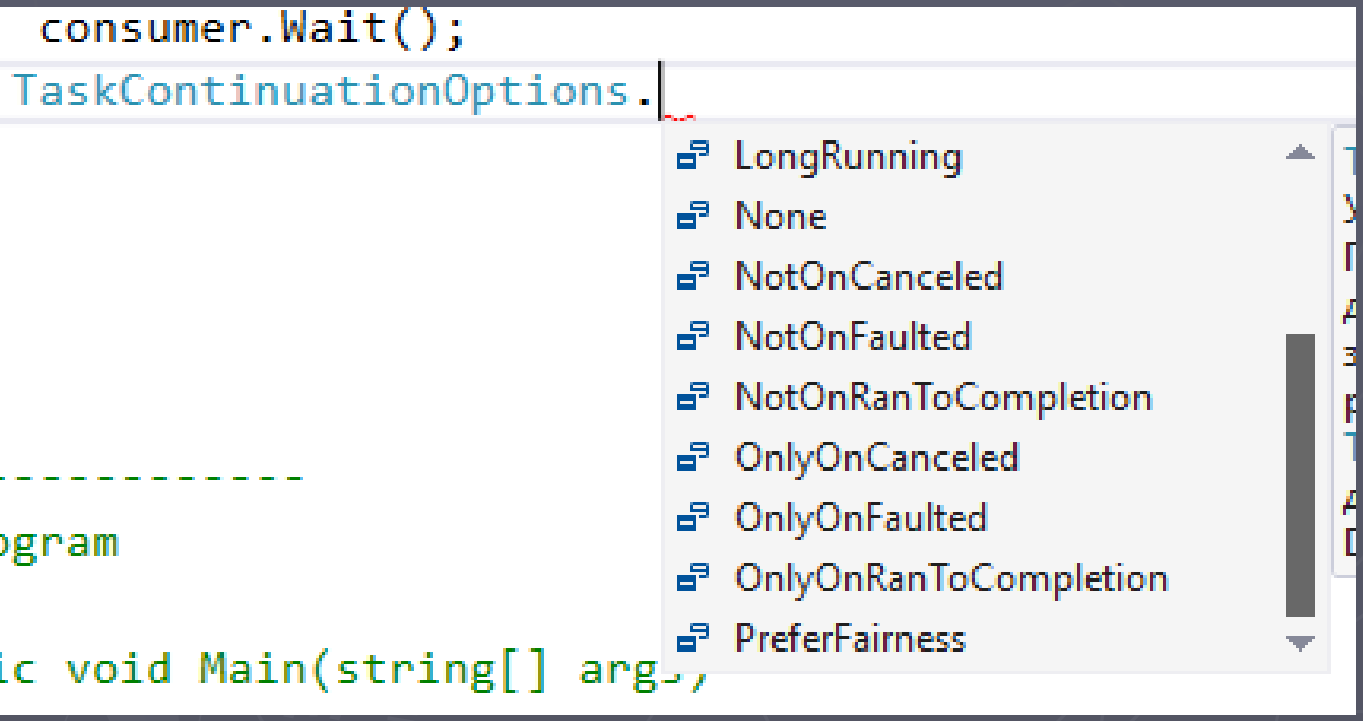
```
Main finished
Doing..continuationД
```

```
Task task8 = Task.Run(() => Console.Write("One...."));

Task task9 = Task.Run(() => Console.Write("Two..."));

Task continuation = Task.WhenAll(task8, task9).
                         ContinueWith(t => Console.Write("Three...."));
```



1)планировка на основе завершения множества предшествующих задач

Установка статуса продолжения



2)использовании объекта ожидания

- Объект ожидания – это любой объект, имеющий методы OnCompleted() и GetResult() и свойство IsCompleted.

```
Task<int> what = Task.Run(() => Enumerable.Range(1, 100000)
                              .Count(n=>(n%2==0)));
// получаем объект продолжения
 var awaiter = what.GetAwaiter();

// что делать после окончания предшественника
awaiter.OnCompleted(() =>          {
// получаем результат вычислений предшественника

            int res = awaiter.GetResult();
            Console.WriteLine(res);
        });
```

...50000

делегат, содержащий код продолжения

По умолчанию – в разных потоках

Параллелизм при императивной обработке данных
Класс Parallel

- System.Threading.Tasks.Parallel позволяет распараллеливать циклы и последовательность блоков кода
- For(), ForEach(), Invoke() – шаблоны (на задачах, поддерж искл. и токен отмены)

являются параллельными аналогами циклов for и foreach

могут принимать аргумент типа ParallelOptions для настройки поведения метода

Parallel.For

Parallel.For(int, int, Action<int>)

указание начального и конечного значения счётчика (типа int или long) и тела цикла в виде объекта делегата

```
Parallel.For(1, 10, z=>
                { int r = 1;
                        for (int y = 1; y <= 10; y++)
                    {
                     r *= z;
                    }
            });
```

```
3
4
6
8
9
2
5
1
7
```

Дополнительные возможности:
Досрочный выход
Пакетная обработка диапазонов
Реализация агрегированных операций

```
Parallel.For(1, 10, (int z, ParallelLoopState pd) =>
                {
                    Console.WriteLine(z);
                    int r = 1;
                    for (int y = 1; y <= 10; y++)
                    {
                        r *= z;
                    }

                });
            Console.WriteLine("Stop");
```

```
3
4
6
8
9
1
2
7
5
Stop
```

Поддерживается императивность – оператор следующий за вызовом метода будет вызван после завершения всех задач

Parallel.ForEach

- ParallelLoopResult ForEach<TSource>
- (IEnumerable<TSource> source,Action<TSource> body)

коллекция, делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции

```
ParallelLoopResult listFact = Parallel.ForEach<int>
                              (new List<int>() { 1, 3, 5, 8 },
                              Factorial);
```

```
Parallel.ForEach(Directory.GetFiles(path, "*.jpg"),
                                  image => Process(image));
```

- **IsCompleted**: определяет, завершилось ли полное выполнение параллельного цикла
- **LowestBreakIteration**: возвращает индекс, на котором произошло прерывание работы цикла

```
ParallelLoopResult result = Parallel.For(1, 10, Factorial);

if (!result.IsCompleted)
        Console.WriteLine("Выполнение цикла завершено на итерации {0}",
                                                result.LowestBreakIteration);
```

Досрочный выход из цикла

- break; - последовательном
- Поиск единственного решения (на неизвестной итерации) Поиск всех решений до неизвестной итерации (условие)
- Stop() - отменяет все не начавшиеся и Break() – выполнить гарантированно с меньшими (даже если не начались)

```
Parallel.For(1, 10, (int z, ParallelLoopState pd) =>
                {
                    int r = 1;
                    for (int y = 1; y <= 10; y++)
                    {
                        if (r == 5) pd.Stop();
                        r *= z;
                    }
                });
```

```
Parallel.For(1, 10, i=>
    Console.WriteLine($"{i}," +
            $"{Task.CurrentId} , " +
  $" {Thread.CurrentThread.ManagedThreadId}"
```

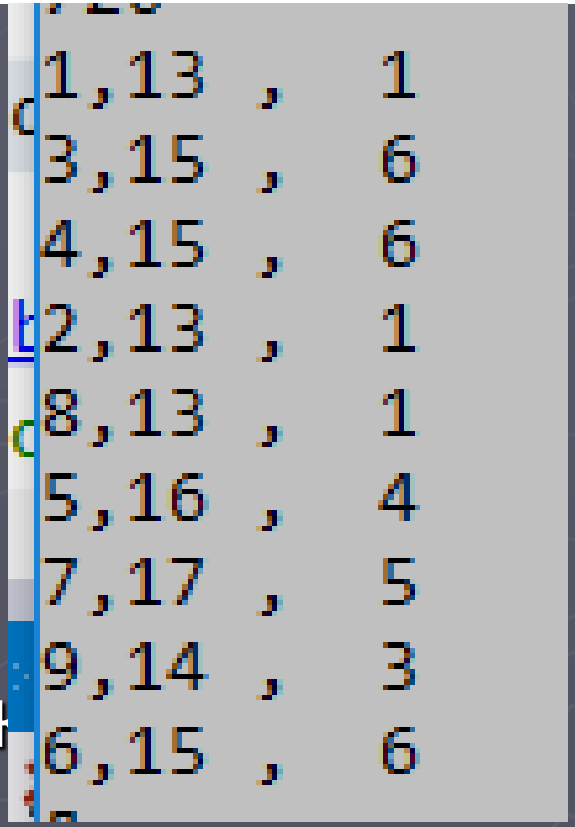
Распределение итераций:

- Равными диапазонами
- - блоками

При стат. декомпоз.

свобод потоки могут

использоваться планировщи



```
Parallel.ForEach(Partitioner.Create(0,5), i =>

Console.WriteLine($"{i}," +
        $"{Task.CurrentId} , "
        $" {Thread.CurrentThread.ManagedThreadId}"));
```

Разделение блоками – сбалансированное разбиение, но требует затрат на синхронизацию доступа к данным

Parallel.Invoke()

- позволяет распараллелить исполнение блоков операторов – набор задач, которые выполняются в одном потоке

Parallel.Invoke (FuncOne, Func Two…)

Методы, лямбда-выражения, массив Action

```
 Parallel.Invoke(
                  () =>     new
WebClient().DownloadFile("http://www.belstu.by", "start.html"),
                  () => new
WebClient().DownloadFile("http://www.go.by", "go.html"));
```

Их можно запустить одновременно

Вызывающий поток должен дождаться завершения всех рабочих элементов

(быстрая сортировка, обработка графов – разделяй и властвуй)

Отмена параллельных операций

```
CancellationTokenSource cancelTokenSource = new
                                        CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;

                Parallel.ForEach<int>(
                    new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8 },
                    new ParallelOptions
                         { CancellationToken = token},
                    Factorial);
        }
```

Коллекции, поддерживающие параллелизм

- System.Collections.Concurrent
- Класс BlockingCollection<T> - реализация шаблона «поставщик - потребитель»

`Add()`	Добавляет элемент в коллекцию
`AddToAny()`	Статический метод, который добавляет элемент в любую из указанных `BlockingCollection<T>`
`CompleteAdding()`	После вызова этого метода добавление элементов невозможно
`GetConsumingEnumerable()`	Возвращает перечислитель, который перебирает элементы с их одновременным удалением из коллекции
`Take()`	Получает элемент и удаляет его из коллекции. Если коллекция пуста и у коллекции был вызван метод `CompleteAdding()`, генерируется исключение
`TakeFromAny()`	Статический метод, который получает элемент из любой указанной `BlockingCollection<T>`
`TryAdd()`	Пытается добавить элемент в коллекцию, в случае успеха возвращает `true`. Дополнительно может быть задан временной интервал и токен отмены
`TryAddToAny()`	Статический метод, который пытается добавить элемент в любую из указанных коллекций

`TryTake()`	Пытается получить элемент (с удалением из коллекции), в случае успеха возвращает `true`
`TryTakeFromAny()`	Статический метод, который пытается получить элемент из любой указанной `BlockingCollection<T>`
`BoundedCapacity`	Свойство возвращает максимальное число элементов, которое можно добавить в коллекцию без блокировки поставщика (данный параметр может быть задан при вызове конструктора `BlockingCollection<T>`)
`IsAddingCompleted`	Возвращает `true`, если вызывался `CompleteAdding()`
`IsCompleted`	Возвращает `true`, если вызывался `CompleteAdding()` и коллекция пуста

```
static int x = 0;
```

поставщик - потребител

```
BlockingCollection<int> blockcoll = new BlockingCollection<int>();
            for (int producer = 0; producer < 5; producer++)
            {
                Task.Factory.StartNew(() =>
                {    x++;
                    for (int ii = 0; ii < 3; ii++)
                    {
                        x++;
                        Thread.Sleep(100);
                        int id =  x;
                        blockcoll.Add(id);
                        Console.WriteLine("Produser  add " + id);
                    }
                });
            }

            Task consumer = Task.Factory.StartNew(
                    () =>
                    {
                        foreach (var item in blockcoll.GetConsumingEnumerable())
                        {
                            Console.WriteLine(" Reading " +item);
                        }
                    });
            consumer.Wait();
```

```
Produser  add 8
Produser  add 8
Produser  add 8
Produser  add 8
Produser  add 12
Produser  add 12
Produser  add 12
Produser  add 15
Produser  add 16
Produser  add 16
Produser  add 18
Produser  add 16
 Reading 8
 Reading 8
 Reading 8
 Reading 8
 Reading 12
 Reading 12
 Reading 12
 Reading 15
 Reading 16
 Reading 16
 Reading 16
 Reading 18
Produser  add 18
 Reading 18
Produser  add 19
 Reading 19
Produser  add 20
 Reading 20
```

Типовые модели параллельных вычислений

- Модель делегирования (упр-раб)
  - fork-join Invoke
- Сеть с равноправными узлами
  - TPL For Foreach PLINQ
- Конвейер
  - Аснихронные задачи LongRunning
- Модель «Производитель -потребитель»

Асинхронное программирование

- При асинхронном вызове поток выполнения разделяется на две части:
- в одной выполняется метод,
- а в другой – процесс программы.

- Асинхронный вызов методов реализуется средой исполнения при помощи пула потоков

Назначение:

- Доступ к web HttpClient
- Работа с файлами
- Работа с изображениями
- WCF программирование

Асинхронные методы, async и await

- .NET Core
- **Task-based Asynchronous Pattern**

обычно является задачей

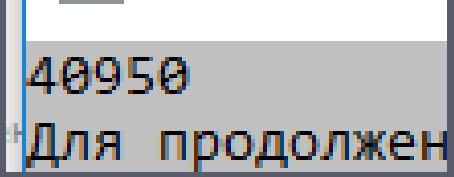
```
var результат = await выражение;
оператор(ы);
```

```
await выражение;
оператор(ы);
```

```
var awaiter = выражение.GetAwaiter();
 awaiter.OnCompleted(()=>
    {
            var результат = awaiter.GetResult();
             оператор(ы);
     }
```

может применяться только внутри метода (или лямбда-выражения) со специальным модификатором async

Метод должен возвращать void либо тип Task или Task<Tresult>



- методы с модификатором async - называются асинхронные функции

создаёт задачу Task<string> для чтения сайта и возвращает управление

```
 static  private async void
        {
            var web = new WebClient();
            var text =
                  await
web.DownloadStringTaskAsync("https://msdn.microsoft.com/ru-ru/");
            Console.WriteLine(text.Length);
        }

    static void Main(string[] args)
        {

            ReadFromWeb();
            Thread.Sleep(3000);
        }
```

приостановить выполнение метода до тех пор, пока эта задача не завершится

асинхронная функция, которая не блокирует при вызове основной поток

- В стандартных классах платформы .NET многие методы, выполняющие долгие операции, получили поддержку в виде асинхронных аналогов. Async в названии

```
Func<Task> someM = async () =>

// Task.Delay() - асинхронный аналог Thread.Sleep()
await Task.Delay(1000);
Console.Write("working ..... ");

};
```

- 1) В сигнатуру метода добавляется async
- 2) Имя метода async, по соглашению, заканчивается суффиксом «Async»
- 3) Тип возврата
  - Task <TResult>, если return
  - Task, если нет оператора return
  - void , если обработчик событий async.
- 4) Обычно метод включает в себя хотя бы одно выражение await, которое отмечает точку, в которой метод не может продолжаться до тех пор, пока ожидаемая асинхронная операция не будет завершена.