Существует **две стратегии** разделения работы между потоками:

- Параллелизм данных (**data parallelism**). Параллелизм данных используется, если необходимо над большим объемом данных выполнить некий набор задач, поэтому этот подход называется параллелизмом данных, поскольку мы разбиваем данные между потоками.
- Параллелизм задач (**task parallelism**). В противоположность параллелизму данных, используется параллелизме задач с распараллеливанием задачи, и в таком случае каждый поток выполняет разную задачу.

*Параллелизм* данных проще и лучше масштабируется на высокопроизводительном оборудовании, поскольку уменьшает или полностью устраняет совместное использование. Кроме того, *параллелизм* данных основывается на том факте, что данных обычно значительно больше, чем отдельных задач, что увеличивает возможности параллельной обработки.

# 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.

Одним из самых главных среди нововведений, внедренных в среду *.NET* Framework 4.0, является ***библиотека распараллеливания задач (TPL)***. Эта библиотека усовершенствует многопоточное *программирование* двумя основными способами:

- Во-первых, упрощает создание и применение многих потоков.
- Во-вторых, позволяет автоматически использовать несколько процессоров.

*TPL* предоставляет возможности для автоматического масштабирования приложений с целью эффективного использования доступных процессоров.

Главной причиной появления нововведений, таких как *TPL* и PLINQ, служит возросшее *значение* параллелизма в современном программировании. Постоянно растет потребность в повышении производительности программ. Библиотека *TPL* определена в пространстве имен *System*.Threading.Tasks . Но для работы с ней обычно требуется также включать в программу *класс System*.Threading, поскольку он поддерживает синхронизацию и другие средства многопоточной обработки, в том числе и те, что входят в *класс* Interlocked.

## Определение многопоточности

***Поток*** (thread) – это управляемая *единица* исполняемого кода. В многозадачной среде, основанной на потоках, у всех работающих процессов обязательно имеется основной *поток*, но их может быть и больше. Это означает, что в одной программе могут выполняться несколько задач асинхронно. К примеру, редактирование текста в текстовом редакторе во время печати, т.к. эти две задачи выполняются в различных потоках.

***Многопоточность*** (multithreading) – это специализированная форма многозадачности (multitasking). В основном, выделяют два типа многозадачности: основанную на процессах (process-based) и основанную на потоках (thread-based).

***Процесс*** (process) – это по сути запущенная *программа*. Следовательно, основанная на процессах *многозадачность* – средство, позволяющее компьютеру выполнять несколько операций (программ) одновременно. Основанная на процессах *многозадачность* предоставляет одновременно редактировать текст в текстовом редакторе и работать с другой запущенной программой.

## Создание вторичных потоков

При создании многопоточного приложения, необходимо, руководствоваться следующими шагами:

1. Создается метод, который будет точкой входа для нового потока.
2. Создается новый делегат `ParametrizedThreadStart` (или `ThreadStart`), передав конструктору метод, который определен на предыдущем шаге.
3. Создается объект **Thread**, передав в качестве аргумента конструктора `ParametrizedThreadStart/ThreadStart`.
4. Устанавливается начальные характеристики потока (имя, приоритет и т.п.).
5. Вызывается метод `Thread.Start()`. Это действие запустит поток на методе, который указан делегатом, созданным на втором шаге, как только это будет возможно.

Всего различают два типа потоков:

- Потоки переднего плана (**foreground threads**) или приоритетный. По умолчанию каждый поток, создаваемый через метод `Thread.Start()`, автоматически становится потоком переднего плана. Данный тип потоков обеспечивают предохранение текущего приложения от завершения. Среда CLR не остановит приложение до тех пор, пока не будут завершены все приоритетные потоки.
- Фоновые потоки (**background threads**). Данный вид потоков называется также потоками-демонами, воспринимаются средой CLR как расширяемые пути выполнения, которые в любой момент времени могут игнорироваться. Таким образом, если все потоки переднего плана прекращаются, то все фоновые потоки автоматически уничтожаются при выгрузке домена приложения. Для создания фоновых потоков необходимо присвоить свойству `IsBackground` значение `true`.

## Задачи и класс Task

В основе библиотеки **TPL** лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов **.NET** задача представлена специальным классом - классом **Task**, который находится в пространстве имен **System.Threading.Tasks**. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

Первый способ.

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

Второй способ заключается в использовании статического метода **Task.Factory.StartNew()**. Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться.

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

Третий способ определения и запуска задач представляет использование статического метода **Task.Run()**. Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

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

# Свойства класса Task

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

- **AsyncState**: возвращает объект состояния задачи
- **CurrentId**: возвращает идентификатор текущей задачи
- **Exception**: возвращает объект исключения, возникшего при выполнении задачи
- **Status**: возвращает статус задачи

# Вложенные задачи

- Одна задача может запускать другую - вложенную задачу. При этом эти задачи выполняются независимо друг от друга. Вложенная задача может завершить выполнение даже после завершения метода Main

- Если необходимо, чтобы вложенная задача выполнялась как часть внешней, необходимо использовать значение **TaskCreationOptions.AttachedToParent.**

- Также как и с потоками, мы можем создать и запустить массив задач. Можно определить все задачи в массиве непосредственно через объект Task

```
Task[] tasks1 = new Task[3]
{
   new Task(() => Console.WriteLine("First Task")),
   new Task(() => Console.WriteLine("Second Task")),
   new Task(() => Console.WriteLine("Third Task"))
};
```

- Задачи продолжения или **continuation task** позволяют определить задачи, которые выполняются после завершения других задач. Благодаря этому можно вызвать после выполнения одной задачи несколько других, определить условия их вызова, передать из предыдущей задачи в следующую некоторые данные.

# Упражнение 1

1. Написать программу в которой создается массив параллельных задач.
Подсчитать параллельно в каждой задаче суммы:

$$\sum_{n=1}^{\infty} \frac{n+1}{2^n} = \frac{2}{2} + \frac{3}{4} + \frac{4}{8} + \ldots + \frac{n+1}{2^n} + \ldots .$$

$$\sum_{n=1}^{\infty} \frac{n+2}{2n-1} = \frac{3}{1} + \frac{4}{3} + \frac{5}{5} + \ldots + \frac{n+2}{2n-1} + \ldots .$$

$$\sum_{n=1}^{\infty} \frac{x^n}{n!} = \frac{x}{1} + \frac{x^2}{1 \cdot 2} + \frac{x^3}{1 \cdot 2 \cdot 3} + \ldots + \frac{x^n}{n!} + \ldots .$$

2. Число итераций не более 1000 или до момента переполнения. Результаты вывести на консоль. Определить в каком порядке будут выполнены задачи.
3. В 3-й последовательности рекомендуется не вычислять факториал, а каждый последующий член суммы вычислять путем деления предыдущего значения последовательности на соответствующее значение **n**.

# Упражнение 2

1. Написать программу с использованием **continuation task**, в которой несколько задач формируют данные, последняя использует эти данные.
2. Первая задача случайным образом выбирает месяц.
3. Вторая задача случайным образом выбирает число месяца с учетом того, какой месяц был выбран в первой задаче.
4. Третья задача случайным образом выбирает время: час, минута, секунда.
5. Пятая задача выводит на консоль – время, число и месяц, полученные в первых задачах.

# Класс Parallel

Класс **Parallel** также является частью TPL и предназначен для упрощения параллельного выполнения кода. Parallel имеет ряд методов, которые позволяют распараллелить задачу.

Одним из методов, позволяющих параллельное выполнение задач, является метод **Invoke**.

```
Parallel.Invoke(Display,
        () => {  Console.WriteLine($"Выполняется задача
{Task.CurrentId}");
                },
        () => Factorial(5));
```

# Parallel.For

Метод Parallel.For позволяет выполнять итерации цикла параллельно.
Он имеет следующее определение:

**For(int, int, Action<int>),**

где первый параметр задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию.

```
Parallel.For(1, 10, Factorial);


static void Factorial(int x)
{  int result = 1;
     for (int i = 1; i <= x; i++) { result *= i;    }
Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Console.WriteLine($"Факториал числа {x} равен {result}");
}
```

## Parallel.ForEach

Метод Parallel.ForEach осуществляет итерацию по коллекции, реализующей интерфейс **IEnumerable**, подобно циклу **foreach**, только осуществляет параллельное выполнение перебора. Он имеет следующее определение:
ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> source, Action<TSource> body),
где первый параметр представляет перебираемую коллекцию, а второй параметр - делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции.

На выходе метод возвращает структуру **ParallelLoopResult**, которая содержит информацию о выполнении цикла.

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

## Выход из цикла

В стандартных циклах **for** и **foreach** предусмотрен преждевременный выход из цикла с помощью оператора **break.** В методах Parallel.ForEach и Parallel.For также можно, не дожидаясь окончания цикла, выйти из него.

```
 ParallelLoopResult result = Parallel.For(1, 10, Factorial);
    if (!result.IsCompleted)
Console.WriteLine($"Выполнение цикла завершено на итерации
{result.LowestBreakIteration}");
}
static void Factorial(int x, ParallelLoopState pls)
{    int result = 1;
    for (int i = 1; i <= x; i++)
    {        result *= i;
        if (i == 5)              pls.Break();
    }
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Console.WriteLine($"Факториал числа {x} равен {result}");
```