

# Асинхронное программирование

Multithreading vs asynchrony

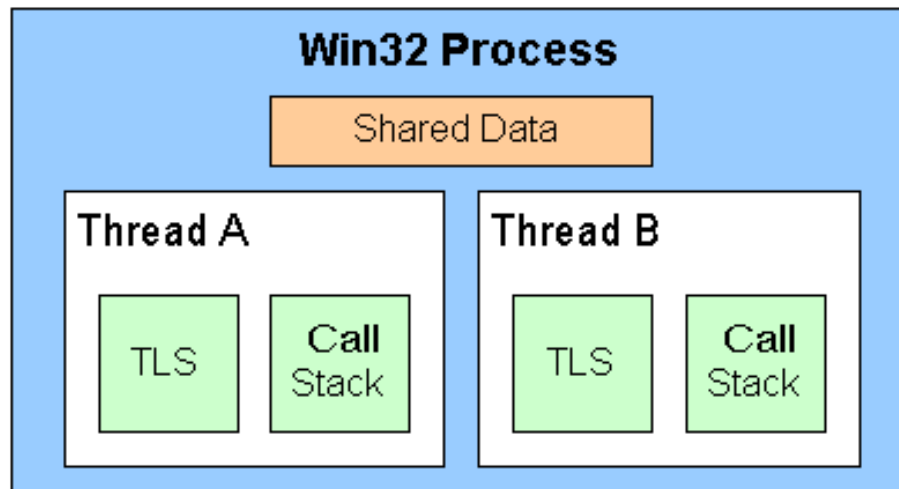
Лекция 7

# Parallel code execution

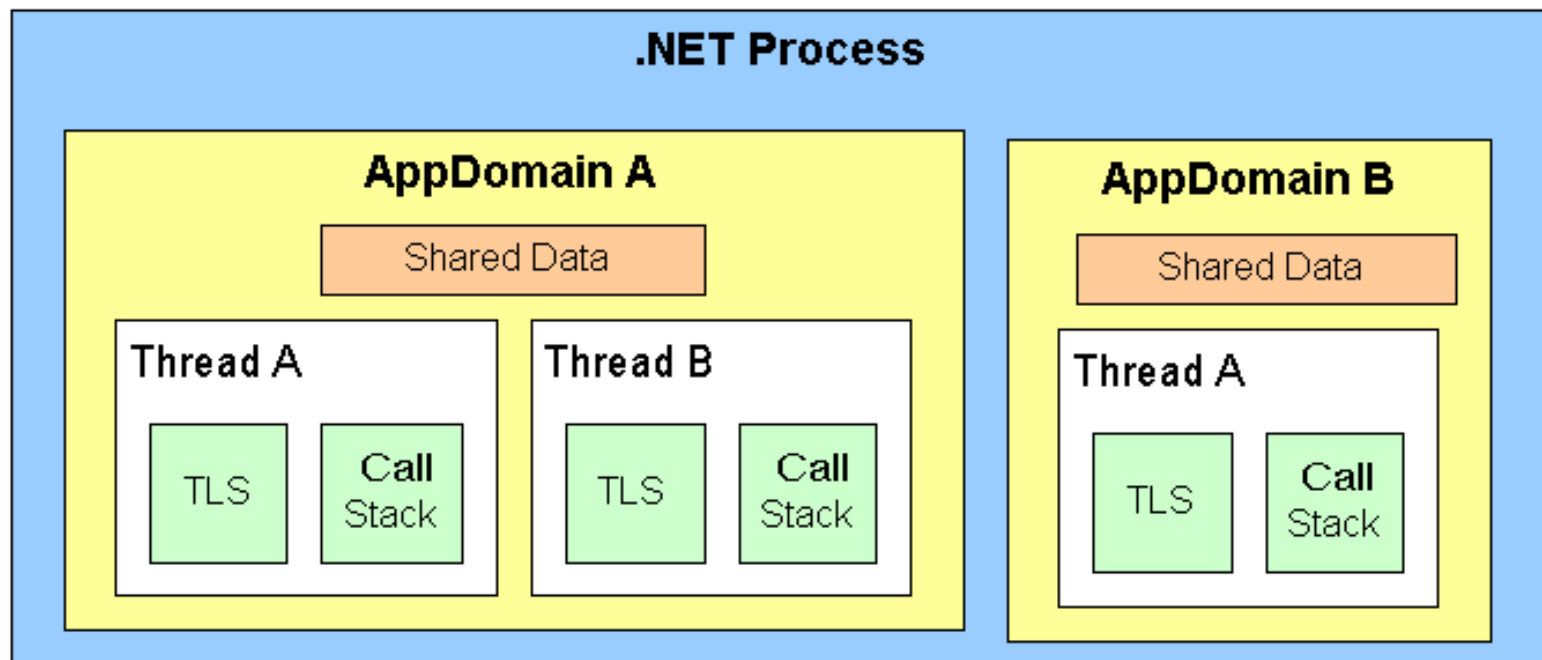


# Процессы и потоки

- **Процесс** – основная программа контейнер, включающая главный поток (**main thread**) и остальные потоки
- **Поток** (нить, thread) – набор инструкций на исполнение для ядра. Поддерживает установку приоритета
- При запуске приложения создается и запускается главный поток
- Любой поток может запускать дополнительные потоки
- Потоки выполняются параллельно и независимо
- Завершение процесса – завершение всех его потоков
- Нет четкой корреляции между потоком операционной системы и управляемым потоком .NET
- Один поток ОС может обслуживать несколько потоков .NET
- Многопоточные приложения могут выполняться и на однопроцессорном компьютере



- **Thread-local storage (TLS)**



# Multithreading vs asynchrony

Понятия много-потокное и асинхронное программирование тесно связаны, но все же между ними есть отличия

Представим аналогию: вы – повар в ресторане(**thread**), вам последовательно поступают заказы(**tasks**). Каждый заказ содержит в себе две операции - варку яиц(**task1**) и выпечку хлеба(**task2**), у вас есть три плиты с таймерами (**resource1, resource2, resource2**).

**synchronous, single-threaded** – вы выбираете плиту, готовите на ней сначала яйца, потом выбираете плиту и готовите на ней хлеб

**synchronous, multi-threaded** – вы готовите сначала яйца, в это время ваш помощник ждет вас и только потом начинает готовить хлеб, события идут последовательно во времени(синхронно)

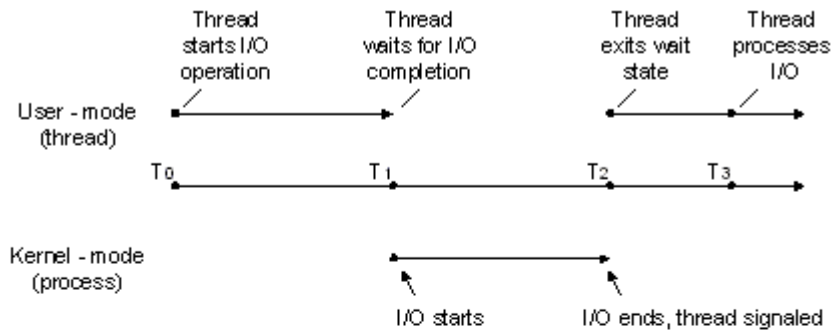
**asynchronous, single-threaded** – вы выбираете плиты и ставите на них хлеб или яйца в любом порядке, заводите таймер для каждого из блюд, пока они готовятся прибираетесь на кухне.

**asynchronous, multi-threaded** - вы нанимаете 2 или более поваров, каждый из них должен выбрать какого блюда не хватает для текущего заказа и дожидаться свободной плиты(**sharing resources**). Вам (**main thread**) приходится координировать их.

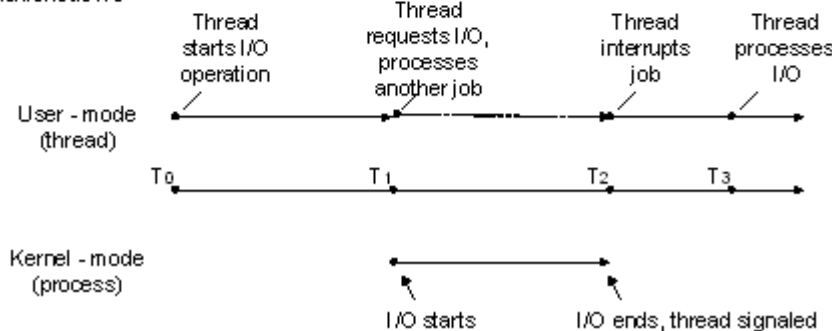
Синхронизация – это координация порядка совершения событий(задач) **во времени**

Многопоточность – это **одновременное** совершение нескольких событий(задач) синхронное или асинхронное по времени относительно **ресурсов**.

Synchronous I/O



Asynchronous I/O





# MULTITHREADING

THREADS ARE NOT GOING TO SYNCHRONIZE THEMSELVES

# Достоинства и недостатки

- При грамотном подходе может значительно ускорить работу приложения (только при многоядерной или много процессорной архитектуре)
- Позволяет повысить отзывчивость пользовательского интерфейса (даже на одном ядре)
- Позволяет ускорить работу приложения за счет одновременного выполнения:
  - долгих удаленных операций (выполняющихся на других компьютерах)
    - Например, запрос к базе данных, к сервису или к интернет ресурсу
  - медленных, но мало затратных операций
    - Например, сохранение или чтение с диска
- Трудности разработки (дороговизна разработки)
  - Разбиение и оптимизация программы для многопоточной работы
  - Синхронизация потоков
  - Тестирование
- Трудности тестирования и отладки
  - Трудно обнаружимые ошибки
  - Невоспроизводимые ошибки
  - Непредсказуемые ошибки
- При неграмотном подходе может замедлить приложение
  - На создание и поддержание работы потоков тратятся ресурсы



# Потоки в .NET

- Пространства имен
  - System.Threading
  - System.Threading.Tasks (.NET 4)
  - System.ComponentModel (поток для UI, BackgroundWorker)
- Класс System.Threading.Thread
  - Методы для работы с потоками
  - Статические члены для текущего потока
  - static Thread Thread.CurrentThread – текущий поток
- Единица кода для запуска в потоке – метод
  - В отдельном потоке всегда запускается кокой-то метод

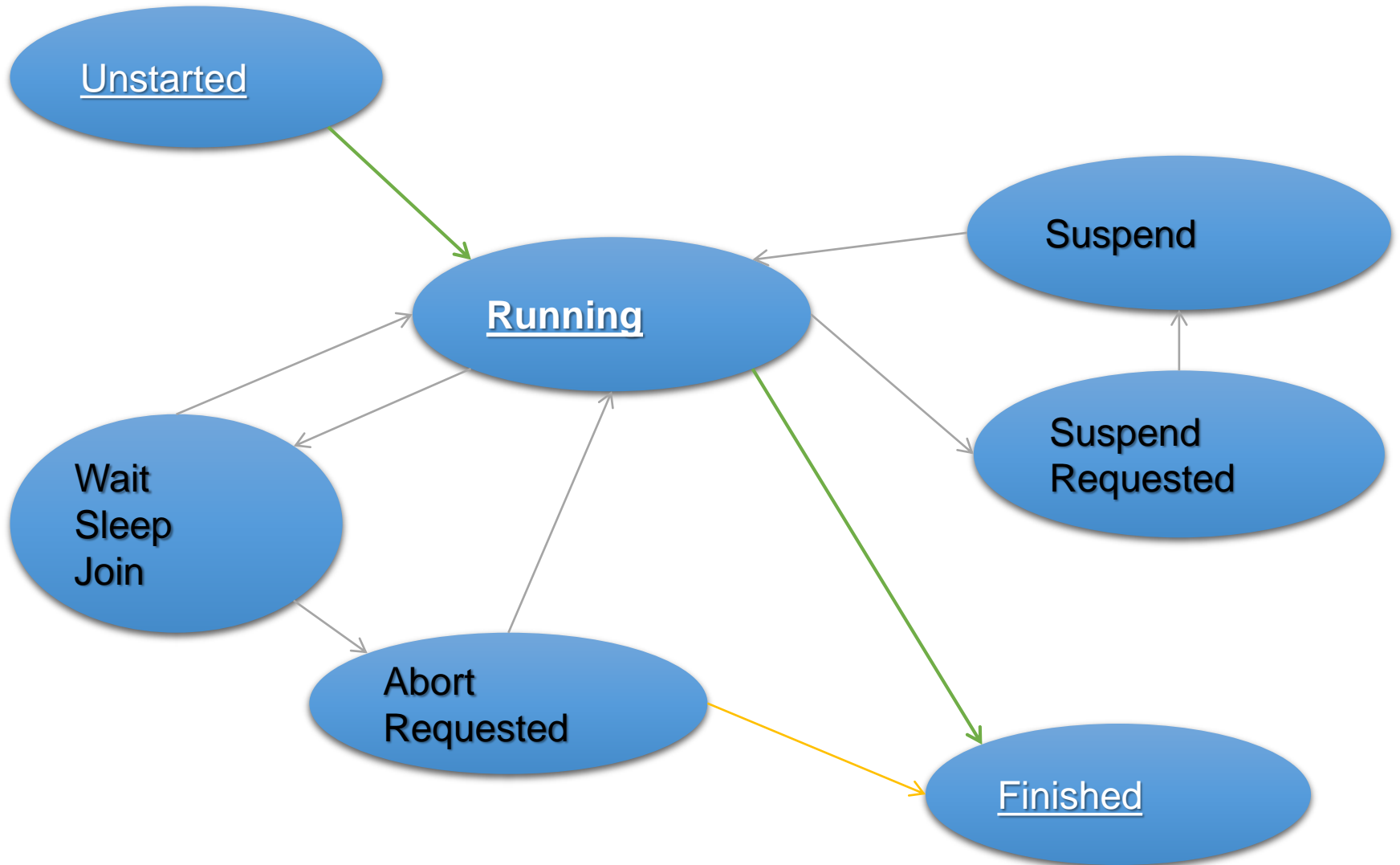
# Класс Thread

- Свойства потока
  - Name – имя потока (удобно использовать для отладки)
  - ManagedThreadId – уникальный ID потока
  - Priority – приоритет потока
  - IsAlive – поток запущен и не приостановлен
  - ThreadState – состояние потока
  - IsBackground – фоновый ли поток
  - IsThreadPoolThread – принадлежит ли поток пулу потоков CLR
- Полезные методы и свойства для работы с потоками
  - Thread.CurrentThread – ссылка на текущий поток (статическое вычисляемое свойство)
  - Thread.Sleep() – заставляет поток ожидать указанное время (статический метод)
  - thread.Join() – заставляет ожидать текущий поток завершения указанного потока.
  - thread.Abort() – заставляет аварийно завершить поток

# Запуск потока

- Необходимо создать метод, который будет выполняться **НОВЫМ ПОТОКОМ**
  - `public static void threadMethod() {...}`
- Создание экземпляра делегата на метод
  - `ThreadStart` – для запуска потока без параметров
  - `ParameterizedThreadStart` – для запуска потока с одним параметром (но параметр `object`)
- Создание потока и передача ему делегата на метод
  - `Thread thread= new Thread(new ThreadStart(threadMethod));`
- Запуск потока `thread.Start();`

# Состояния потоков



# Завершение потока

- Поток завершится при выходе из метода
- `thread.Abort()` – аварийное завершение потока
  - При этом у прерываемого потока возникает исключение `ThreadAbortedException`
  - Прерываемый поток может обработать исключение `ThreadAbortedException`, но после этого исключение будет вызвано снова
- `thread.AbortReset()` – отмена прерывания потока (если успеть, пока поток еще аварийно не завершился)
- `thread.Join()` – блокировка текущего потока до завершения другого потока
- Завершенный поток нельзя запустить снова

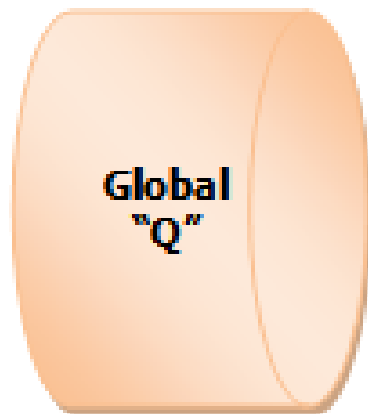
# Фоновые потоки

- Потоки
  - Потоки переднего плана (по умолчанию)
  - Фоновые потоки
- Процесс не завершится пока есть работающие потоки переднего плана
- Фоновые потоки при завершении основного потока получают исключение `ThreadAbortedException` и будут завершены
- Необходима реализация безопасного завершения фонового потока
- Установка потока как фонового  
`thread.IsBackground = true;`

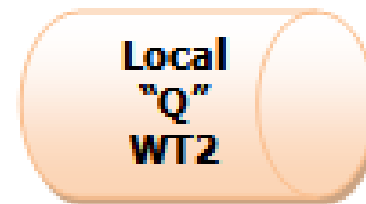
# Пул потоков

- В среде выполнения уже существует несколько запущенных потоков – пул потоков
- Количество поток связано с количеством процессоров.
- При использовании потока из пула потоков нет накладных расходов на создание потока
- В пуле потоки фоновые
- Класс **ThreadPool** – позволяет получить доступ к пулу потоков .NET
- Постановка задания в очередь
  - Создание экземпляра делегата `void WaitCallback(object state )`
  - Постановка в очередь `ThreadPool.QueueUserWorkItem`
  - `(new WaitCallback(threadMethod), obj);`
- Переданное задание уже нельзя отменить

## Thread Pool



Worker  
Thread 1



Worker  
Thread 2

Task Scheduler

Program  
Thread



# Порядок выполнения потоков непредсказуем

- Потоки выполняются параллельно и независимо. Нельзя предсказать очередность выполнения блоков кода потоками.

```
static void Main()
{
    Thread t = new Thread(Write1);
    t.Start();
    while (true) Console.Write("-"); // Все время печатать '-'
}
```

```
static void Write1()
{
    while (true) Console.Write("1"); // Все время печатать '1'
}
```

# Операции не являются атомарными

```
class increment {  
    decimal l = 0;  
    public void inc() {  
        for (int i = 0; i < 100000; ++i) l = l + 1;  
        Console.WriteLine(l);  
    } }  

```

```
class Program {  
    static void Main(string[] args) {  
        increment i = new increment();  
        for (int j = 0; j < 10; ++j)  
            new Thread(i.inc).Start();  
    }  
}
```

# Синхронизация потоков

- С помощью класса Interlocked
- Конструкция lock
- Класс Monitor
- Классы ReaderWriterLock, ReaderWriterLockSlim
- Класс Mutex
- Семафоры
- EventWaitHandle

# Класс Interlocked

- Атомарные операции. Статические члены
  - `Interlocked.Increment(ref i);`  $i$  – long или int
  - `Interlocked.Decrement(ref i);`  $i$  – long или int
  - `Interlocked.Add(ref i1, i2);` Переменные int, long
  - `Interlocked.Exchange(ref i, value);`
  - `Interlocked.Exchange<T>(ref T i, T value);`
  - `Interlocked.CompareExchange(ref i, value, compared);`
    - Если  $i == compared$ , то  $i = value$ . Переменные типов: int, long, float, double, object
  - `Interlocked.CompareExchange <T> (ref T i, T value, T compared)` – для ссылочных типов

# Конструкция lock

- Необходимо определить единую доступную всем потокам ссылочную переменную
- Если объект в переменной не блокирован, то поток проходит беспрепятственно через оператор lock, блокируя объект
- Если объект в переменной блокирован, то поток остановится на операторе lock и будет ожидать пока другой поток не выйдет из конструкции lock
- Например:

```
public object lockObject = new object();
```

```
lock (lockObject)  
{  
    // Операции с разделяемыми ресурсами  
}
```

# Необходимо

- Как можно быстрее освобождать блокировку
- Избегать взаимоблокировок

```
lock (A)
```

```
{
```

```
    lock (B)
```

```
    {
```

```
    }
```

```
}
```

```
lock (B)
```

```
{
```

```
    lock (A)
```

```
    {
```

```
    }
```

```
}
```

- Блокировать только ссылочную переменную
- Экземпляр объекта должен быть один и тот же для всех потоков

# Асинхронное программирование

Делегаты, Task Parallel  
Library (TPL)

Лекция 7

# Асинхронный вызов методов в делегате

- Любой делегат имеет помимо метода для синхронного вызова – `Invoke()`, методы для асинхронного вызова `BeginInvoke()`, `EndInvoke()`
- Пусть есть делегат вида `res f(args)`
- Тогда:
  - `IAsyncResult f.BeginInvoke(args, AsyncCallback callback, object obj)` – начинает вызов и передает параметры `args`
  - `res f.EndInvoke(IAsyncResult ires)` – ожидает завершения и возвращает значение
- `AsyncCallback callback` – делегат будет вызван при окончании вычисления



# Интерфейс IAsyncResult

- Свойство `bool IsCompleted` – завершено ли вычисление
- Свойство `object AsyncState` – позволяет передавать параметры для последующей идентификации вызванного метода

# Класс Task

- Простой запуск выполнения делегата в ThreadPool
  - `void inc() {}`
  - `Task t = new Task(inc);`
  - `t.Start();`
- Быстрый старт заданий
  - `Task t = Task.Factory.StartNew(inc);`
- Ожидание завершения `t.Wait();`
- Продолжение выполнения
  - `void a(Task t) { }`
  - `t.ContinueWith(a);`

# Асинхронные методы, **async** и **await**

- В .NET 4.5 во фреймворк были добавлены два новых ключевых слова **async** и **await**, цель которых - упростить написание асинхронного кода.
- Операторы **async** и **await** используются вместе для создания асинхронного метода.
- Ключевое слово **async** указывает, что метод или лямбда-выражение являются асинхронными.
- Оператор **await** применяется к задаче в асинхронных методах, чтобы приостановить выполнение метода до тех пор, пока эта задача не завершится. Выполнение потока, в котором был вызван асинхронный метод, не прерывается

# Пример использования

- ```
static void Main(string[] args)
{
    DisplayResultAsync();
    Console.ReadLine();
}

static async void DisplayResultAsync()
{
    int num = 5;

    int result = await Task.Run(() =>
    {
        int res = 1;
        for (int i = 1; i <= num; i++)
        {
            res *= i;
        }
        return res;
    });
    Thread.Sleep(3000);
    Console.WriteLine("Факториал числа {0} равен {1}", num, result);
}
```

# Работа с параллельными коллекциями

# Работа с коллекциями

- Некоторые коллекции содержат объект для синхронизации (для использования с lock) – `SyncRoot`

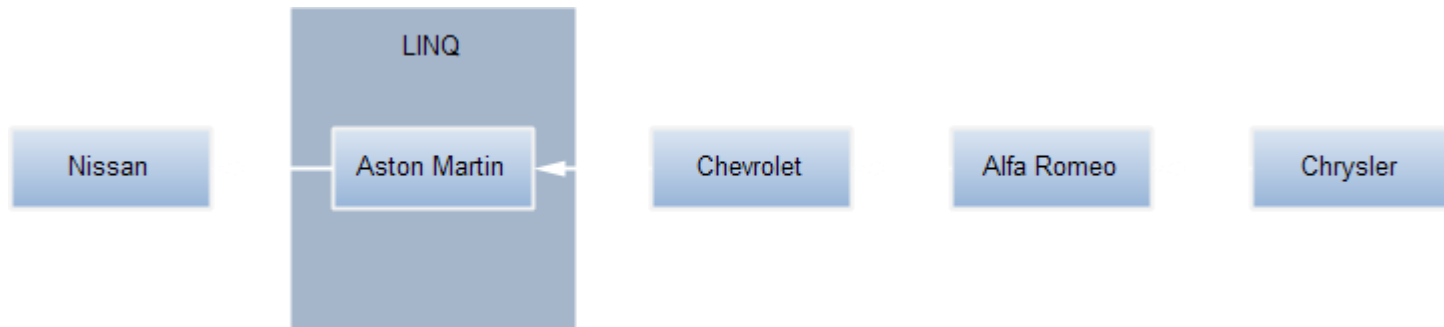
```
int[] col = new int[2];  
  
.....  
lock(col.SyncRoot)  
{  
    // работа с массивом  
}
```

- Имеются специальные коллекции, доступ к которым из разных потоков не требует синхронизации, поскольку они содержат внутренние механизмы синхронизации
  - `ConcurrentQueue<T>` - очередь
  - `ConcurrentStack<T>` - стек
  - `ConcurrentDictionary<TKey, TValue>` - словарь
  - `ConcurrentBag<T>` - простой список

# Класс Parallel

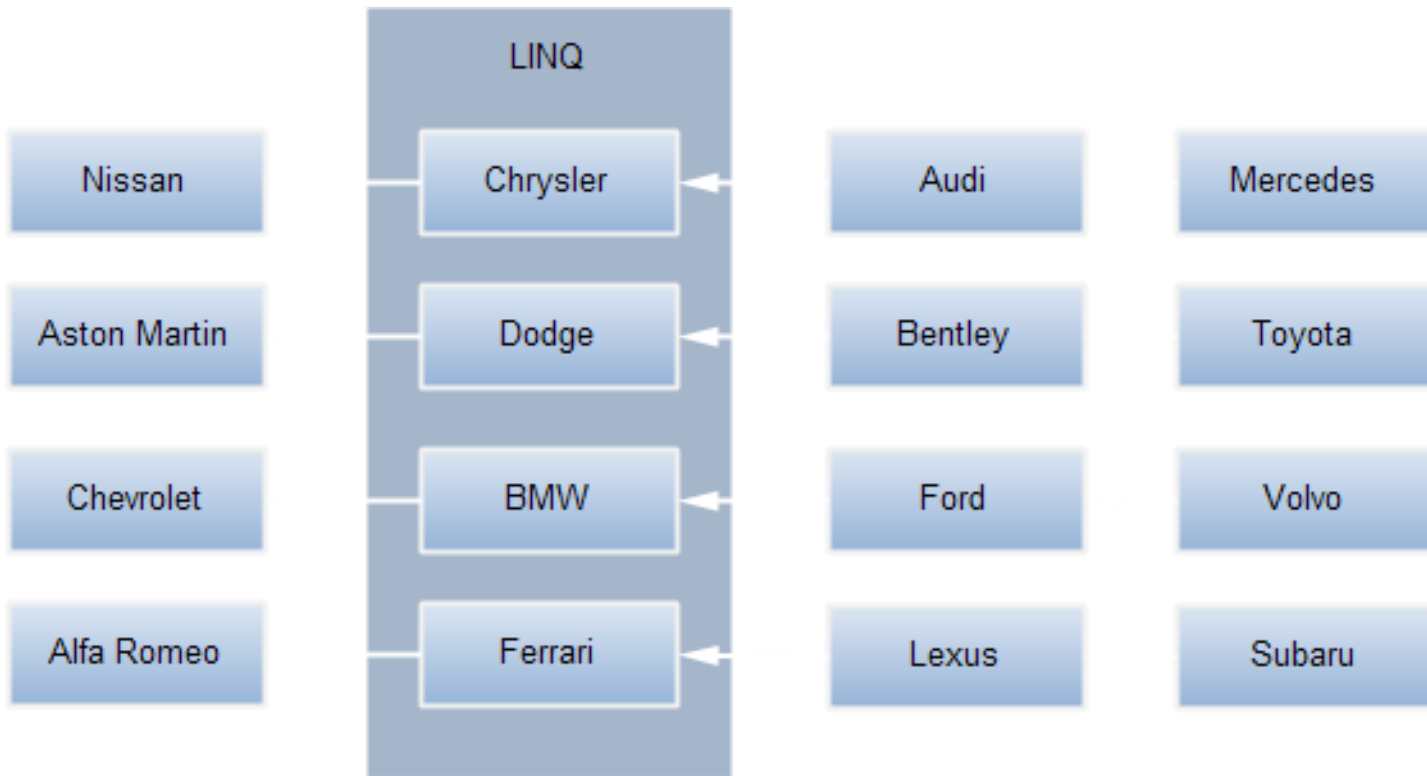
- **Parallel.For**(initvalue, endvalue, Action<T>); - Выполнение цикла в максимально возможном числе потоков (ThreadPool). В цикле выполняется делегат Action<T> (который принимает 1 параметр T и, ничего не возвращает). Числом потоков управляет CLR
- **Parallel.ForEach**<T>(IEnumerable<T>, Action<T>); - Выполнение делегата Action<T> над всеми элементами перечисления в максимально возможном числе потоков. Числом потоков управляет CLR
  - List<int> l = new List<int>();
  - public void dec(int i) {}
  - Parallel.For(0, 10, dec);
  - Parallel.ForEach<int>(l, dec);
  - Parallel.ForEach(l, dec);
- **Parallel.Invoke**(params Action[] actions) – выполнение делегатов в отдельных потоках, если возможно
  - Parallel.Invoke(Print, PrintToScreen, SendToEmail, () => Console.WriteLine("Печатаем"));
- Класс **ParallelOptions** может использоваться для подстройки операций Parallel
  - *MaxDegreeOfParallelism* – ограничивает максимально число одновременно выполняющихся задач в классе Parallel.
  - *CancellationToken* – позволяет отменять задания, выполняющиеся классом Parallel

```
string[] cars = { "Nissan", "Aston Martin",  
"Chevrolet", "Alfa Romeo", "Chrysler", "Dodge",  
"BMW", "Ferrari", "Audi", "Bentley", "Ford", "Lexus",  
"Mercedes", "Toyota", "Volvo", "Subaru", "Жигули"};  
  
string auto = cars.Where(p =>  
p.StartsWith("S")).First(); Console.WriteLine(auto)
```





```
string auto = cars.AsParallel()  
.Where(p => p.StartsWith("S")).First();  
Console.WriteLine(auto);
```



```
using System.Diagnostics;
```

```
IEnumerable<int> nums1 = Enumerable.Range(0, Int32.MaxValue);
```

```
Stopwatch sw = Stopwatch.StartNew();
```

```
int sum1 = (from n in nums1 where n % 2 == 0 select n).Count();
```

```
Console.WriteLine("Результат последовательного выполнения: " + sum1
```

```
+ "\nВремя: " + sw.ElapsedMilliseconds + " мс\n");
```

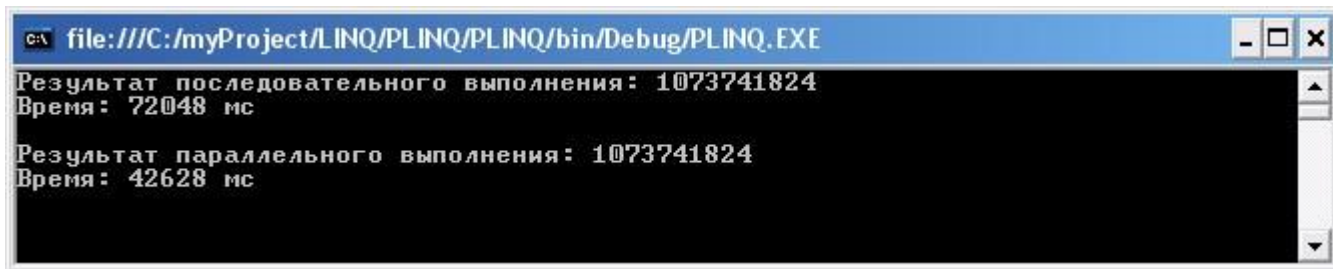
```
ParallelEnumerable.Range(0, Int32.MaxValue);
```

```
sw.Restart();
```

```
int sum2 = (from n in nums2.AsParallel() where n % 2 == 0 select n).Count();
```

```
Console.WriteLine("Результат параллельного выполнения: " + sum2
```

```
+ "\nВремя: " + sw.ElapsedMilliseconds + " мс");
```



```
file:///C:/myProject/LINQ/PLINQ/PLINQ/bin/Debug/PLINQ.EXE
Результат последовательного выполнения: 1073741824
Время: 72048 мс
Результат параллельного выполнения: 1073741824
Время: 42628 мс
```