

Введение в многопоточное программирование

БГУ, ММФ, кафедра веб-технологий и компьютерного
моделирования

Автор: Кравчук Анжелика Ивановна

МНОГОПОТОЧНОСТЬ. НАЧАЛО РАБОТЫ



Распространенные сценарии применения параллелизма

Большинству приложений приходится иметь дело с более чем одной активностью, происходящей одновременно (параллелизм)!

Распространенные сценарии применения параллелизма

- Написание отзывчивых пользовательских интерфейсов
- Обеспечение одновременной обработки запросов (ASP.NET, WCF)
- Параллельное программирование

Общий механизм, с помощью которого программа может выполнять код одновременно, называется **многопоточностью**

Обзор и ключевые понятия. Потоки vs. процессы

Поток – это путь выполнения, который может проходить независимо от других

Все потоки одного приложения логически содержатся в пределах процесса – модуля операционной системы, который представляет изолированную среду с предоставленным отдельным квантом процессорного времени, в котором исполняется приложение

В некоторых аспектах потоки и процессы схожи – например, время разделяется между процессами, исполняющимися на одном компьютере, так же, как между потоками одного приложения

Ключевое различие состоит в том, что процессы полностью изолированы друг от друга. Потоки разделяют память (кучу) с другими потоками этого же приложения. Благодаря этому один поток может поставлять данные в фоновом режиме, а другой – показывать эти данные по мере их поступления

Одиночный процесс Windows

Разделяемые данные

Поток А

TLS

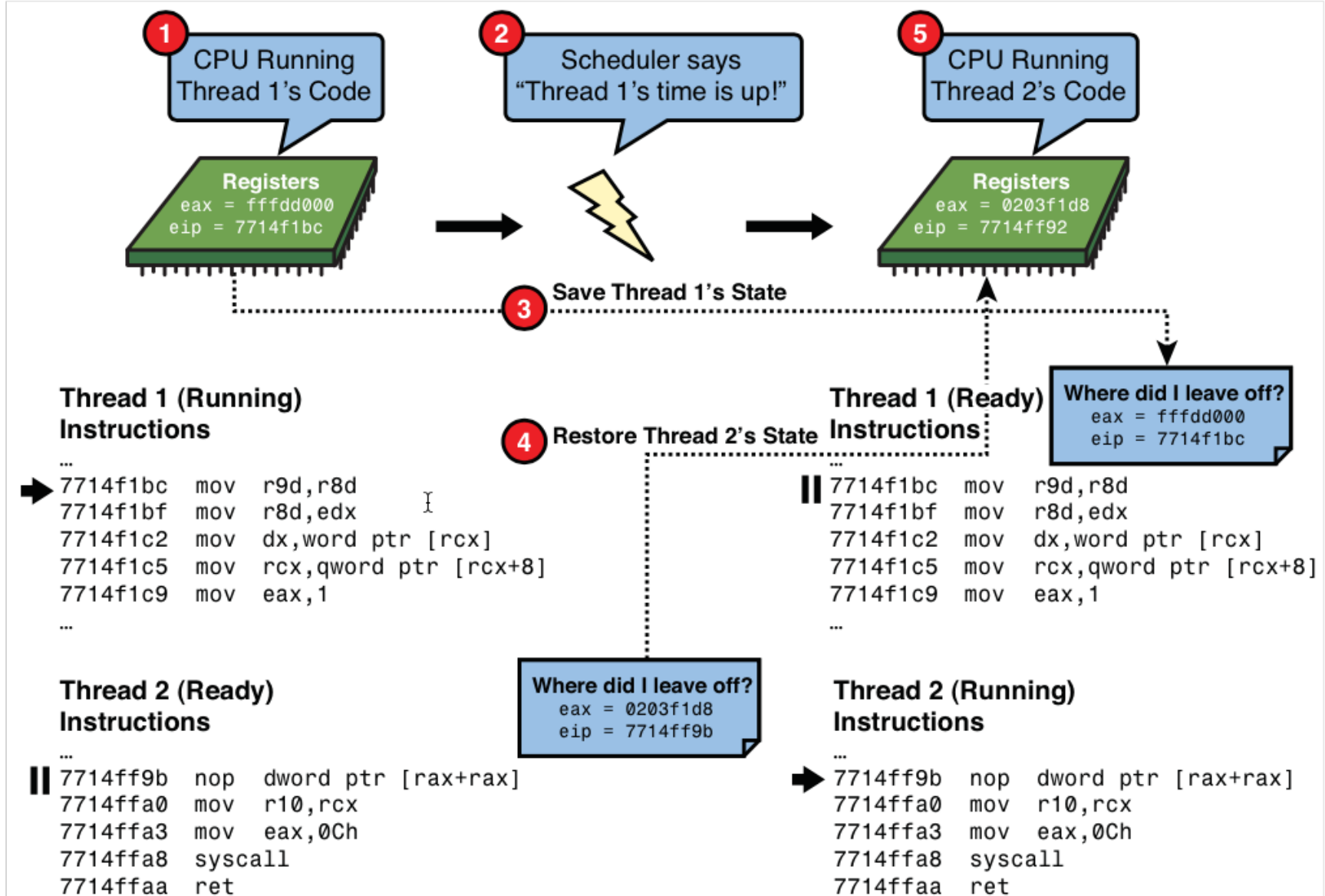
Стек
ВЫЗОВОВ

Поток В

TLS

Стек
ВЫЗОВОВ

Обзор и ключевые понятия



Обзор и ключевые понятия

Управление многопоточностью осуществляет планировщик потоков, эту функцию CLR обычно делегирует операционной системе

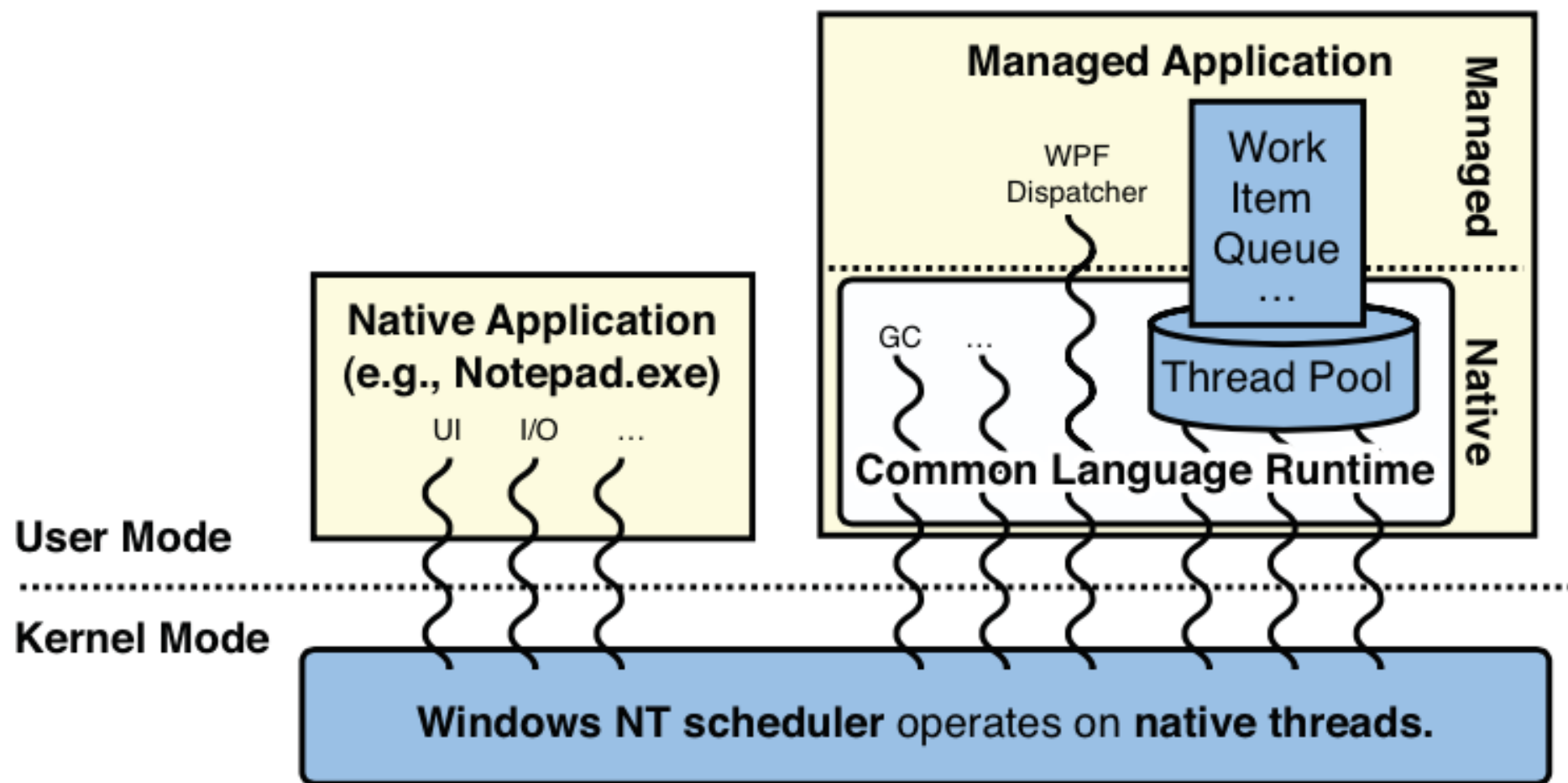
Планировщик потоков гарантирует, что активным потокам выделяется соответствующее время на выполнение, а потоки, ожидающие или заблокированные, к примеру, на ожидании блокировки, или пользовательского ввода – не потребляют времени CPU

На однопроцессорных компьютерах планировщик потоков использует квантование времени – быстрое переключение между выполнением каждого из активных потоков, что приводит к непредсказуемому поведению

Типичное значение кванта времени – десятки миллисекунд (20-30 миллисекунд в случае Windows) – выбрано как намного большее, чем затраты CPU на переключение контекста между потоками (несколько микросекунд)

На многопроцессорных компьютерах многопоточность реализована как смесь квантования времени и подлинного параллелизма, когда разные потоки выполняют код на разных CPU

Контекст потока - структура, создаваемая Windows для каждого потока, служащая, в частности, для запоминания состояния регистров на момент окончания последнего по времени кванта исполнения потока



Недостатки

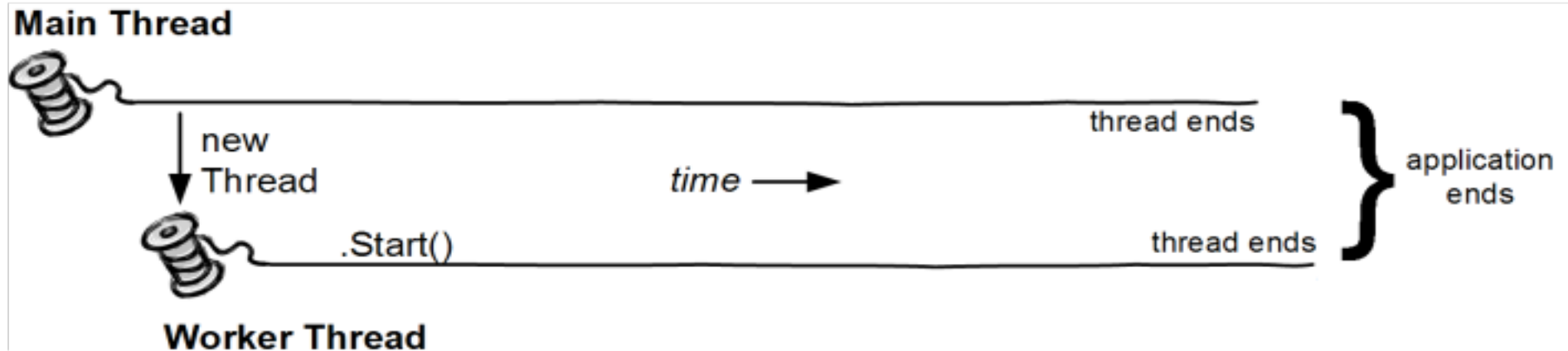
- значительное увеличение сложности программ - необходимость организации взаимодействия потоков - зависимость продолжительность цикла разработки, а также количество спорадически проявляющихся и трудноуловимых ошибок в программе
- отнимает ресурсы и время CPU на создание потоков и переключение между потоками - когда используются операции чтения/записи на диск, более быстрым может оказаться последовательное выполнение задач в одном или двух потоках, чем одновременное их выполнение в нескольких потоках

Обзор и ключевые понятия

C# поддерживает параллельное выполнение кода через многопоточность

Поток (thread, нить) – это независимый путь исполнения, способный выполняться одновременно с другими потоками

Программа на C# запускается как единственный поток, автоматически создаваемый CLR и операционной системой («главный» поток), и становится многопоточной при помощи создания дополнительных потоков



C#-приложение можно сделать многопоточным двумя способами: либо явно создавая дополнительные потоки и управляя ими, либо используя возможности неявного создания потоков .NET Framework – BackgroundWorker, пул потоков, потоковый таймер, Web-службы или приложение ASP.NET

Создание и запуск потоков

Основные классы, предназначенные для поддержки многопоточности, сосредоточены в пространстве имен `System.Threading`. На платформе .NET каждый поток выполнения представлен объектом класса `Thread`. Для организации собственного потока необходимо создать объект этого класса

```
public Thread(ThreadStart start);  
public Thread(ThreadStart start, int maxStackSize);  
public Thread(ParameterizedThreadStart start);  
public Thread(ParameterizedThreadStart start, int maxStackSize);
```

Создание и запуск потоков

В качестве первого параметра конструктору передается делегат, инкапсулирующий метод, выполняемый в потоке

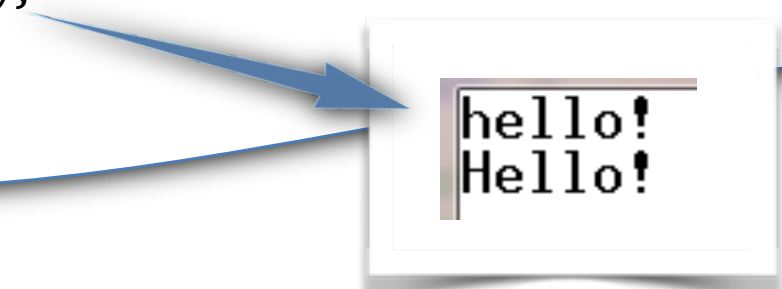
```
public delegate void ThreadStart();  
public delegate void ParameterizedThreadStart(object obj);
```

```
Thread t = new Thread(new ThreadStart(Go));  
Thread t = new Thread(Go);  
Thread t = new Thread(delegate()  
    {  
        Console.WriteLine("Hello!");  
    });  
Thread t = new Thread(() => Console.WriteLine("Hello!"));
```

Создание и запуск потоков

Вызов метода **Start** начинает выполнение потока. Поток продолжается до выхода из исполняемого метода

```
. . .  
static void Main(string[] args)  
{  
    Thread t = new Thread(() => Console.WriteLine("Hello!"));  
    t.Start();  
    Go();  
}  
  
static void Go()  
{  
    Console.WriteLine("hello!");  
}  
. . .
```




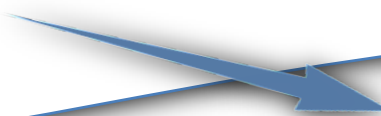
A blue arrow points from the `Console.WriteLine("hello!");` line in the `Go()` method to a small window icon. Another blue arrow points from the `Console.WriteLine("Hello!");` line in the `Main` method to the same window icon. The window icon displays the output of the program.

```
hello!  
Hello!
```

Создание и запуск потоков

Передача данных в ThreadStart

```
. . .  
static void Main(string[] args)  
{  
    Thread t = new Thread(Go);  
    t.Start(true);  
    Go(false);  
}  
  
static void Go(object upperCase)  
{  
    bool upper = (bool)upperCase;  
    Console.WriteLine(upper ? "HELLO!" : "hello!");  
}  
. . .
```




hello!
Hello!

Создание и запуск потоков

Передача данных в ThreadStart

```
static void Main()
{
    string text = "Hello!";
    Thread t = new Thread(() => WriteText(text));
    t.Start();
}

static void WriteText(string text) { Console.WriteLine(text); }
```



Удобство лямбда-выражения (анонимного метода) состоит в том, что нужный метод можно вызвать с любым количеством аргументов и безо всякого приведения типа

■ Thread - свойства

ApartmentState - свойство

CurrentContext - свойство

CurrentCulture - свойство

CurrentPrincipal - свойство

CurrentThread - свойство

CurrentUICulture - свойство

ExecutionContext - свойство

IsAlive - свойство

IsBackground - свойство

IsThreadPoolThread - свойство

ManagedThreadId - свойство

Name - свойство

Priority - свойство

ThreadState - свойство

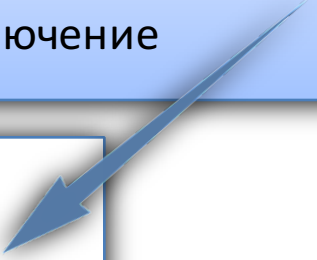
Создание и запуск потоков. Свойства класса Thread

- Статическое свойство `CurrentThread` возвращает объект, представляющий текущий поток
- Свойство `Name` служит для назначения потоку имени
- Целочисленное свойство для чтения `ManagedThreadId` возвращает уникальный числовой идентификатор управляемого потока
- Свойство для чтения `ThreadState`, значением которого являются элементы одноименного перечисления, позволяет получить текущее состояние потока
- Булево свойство для чтения `IsAlive` позволяет определить, выполняется ли поток
- Свойство `Priority` управляет приоритетом выполнения потока относительно текущего процесса. Значением этого свойства являются элементы перечисления `ThreadPriority`: `Lowest`, `BelowNormal`, `Normal`, `AboveNormal`, `Highest`
- Булево свойство `IsBackground` позволяет сделать поток фоновым. Среда исполнения .NET разделяет все потоки на фоновые и основные. Процесс не может завершиться, пока не завершены все его основные потоки. В то же время, завершение процесса автоматически останавливает все фоновые потоки
- Свойства `CurrentCulture` и `CurrentUICulture` имеют тип `CultureInfo` и задают текущую языковую культуру

Создание и запуск потоков. Свойства класса Thread

Поток можно именовать, используя свойство **Name**. Это удобно при отладке: имена потоков можно вывести в **Console.WriteLine** и увидеть в окне *Debug – Threads* в Microsoft Visual Studio. Имя потоку может быть назначено в любой момент, но только один раз – при попытке изменить его будет сгенерировано исключение

```
Thread worker = new Thread(Go);  
worker.Name = "worker";
```




```
Thread worker = new Thread(Go) {Name = "worker"};
```

```
var th = new Thread(DoSomeWork)  
{  
    Name = "Example Thread",  
    Priority = ThreadPriority.BelowNormal,  
    IsBackground = true,  
    CurrentCulture = CultureInfo.GetCultureInfo("ru-RU")  
};
```

Создание и запуск потоков. Основные и фоновые потоки (Foreground and Background Threads)

По умолчанию потоки создаются как основные (foreground), что означает, что приложение не будет завершено, пока один из таких потоков будет исполняться. С# также поддерживает фоновые потоки, они не продлевают жизнь приложению, а завершаются сразу же, как только все основные потоки будут завершены

Статус потока переключается с основного на фоновый при помощи свойства **IsBackground**



```
static void Main(string[] args)
{
    Thread worker = new Thread(delegate() { Console.ReadLine(); });
    if (args.Length > 0)
        worker.IsBackground = true;

    worker.Start();
}
```

Создание и запуск потоков. Приоритеты потоков

Свойство **Priority** определяет, сколько времени на исполнение будет выделено потоку относительно других потоков того же процесса. Существует 5 градаций приоритета потока

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }
```

Значение приоритета становится существенным, когда одновременно исполняются несколько потоков

```
worker.Priority = ThreadPriority.Highest;
```

```
System.Diagnostics
```

```
Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;
```

```
Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.Realtime;
```

Создание и запуск потоков. Обработка исключений

Обрамление кода создания и запуска потока блоками try/catch/finally имеет мало смысла

```
static void Main(string[] args)
{
    try
    {
        new Thread(Go).Start();
    }
    catch (Exception ex)
    {
        // Сюда мы никогда не попадем!
        Console.WriteLine("Исключение!");
    }
}

static void Go() { throw new NullReferenceException(); }
```

Создание и запуск потоков. Обработка исключений

Необработанное исключение в любом потоке приводит к закрытию всего приложения

```
static void Main(string[] args)
{
    new Thread(Go).Start();
}

static void Go()
{
    try
    {
        throw new NullReferenceException
    }
    catch (Exception ex)
    {
        //Логирование исключения и/или сигнал другим потокам
    }
}
```

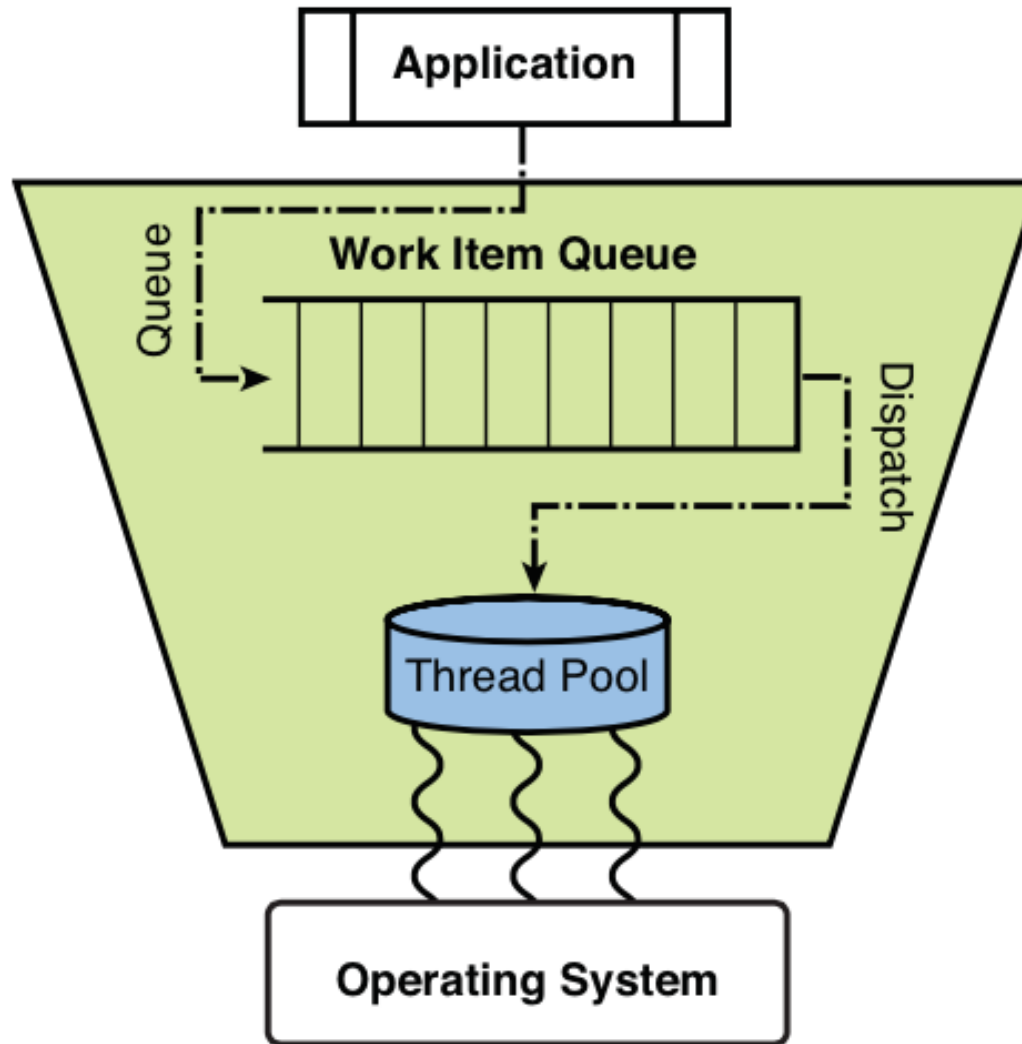
Пул потоков CLR

Создание потоков требует времени. Если есть различные короткие задачи, подлежащие выполнению, можно создать набор потоков заранее и затем просто отправлять соответствующие запросы, когда наступает очередь для их выполнения

Управления списком потоков - класс `ThreadPool` - уменьшает и увеличивает количество потоков в пуле до максимально допустимого. Значение максимально допустимого количества потоков в пуле может изменяться

Двухядерный ЦП - по умолчанию составляет 1023 рабочих потоков и 1000 потоков ввода-вывода

Можно указывать минимальное количество потоков, которые должны запускаться сразу после создания пула, и максимальное количество потоков, доступных в пуле. Если остались какие-то подлежащие обработке задания, а максимальное количество потоков в пуле уже достигнуто, то более новые задания будут помещаться в очередь и там ожидать, пока какой-то из потоков завершит свою работу



Пул потоков CLR

Запросить поток из пула для обработки вызова метода - метод `QueueUserWorkItem` (в дополнение к экземпляру делегата `WaitCallback` позволяет указывать необязательный параметр `System.Object` для специальных данных состояния)

- `QueueUserWorkItem` - метод

`QueueUserWorkItem` - метод (`WaitCallback`)

`QueueUserWorkItem` - метод (`WaitCallback, Object`)

Эти методы ставят «рабочий элемент» вместе с дополнительными данными состояния в очередь пула потоков и сразу возвращают управление приложению. Рабочим элементом называется указанный в параметре `callback` метод, который будет вызван потоком из пула. Этому методу можно передать один параметр через аргумент `state` (данные состояния)

Пул потоков CLR

```
int nWorkerThreads;
int nCompletionThreads;
ThreadPool.GetMaxThreads(out nWorkerThreads, out nCompletionThreads);
Console.WriteLine("Максимальное количество потоков: {0} \nПотоков ввода-вывода доступно: {1}", nWorkerThreads, nCompletionThreads);

for (int i = 0; i < 5; i++)
    ThreadPool.QueueUserWorkItem(JobForAThread);
Thread.Sleep(5000);
```

```
static void JobForAThread(object state)
{
    for (int i = 0; i < 3; i++)
    {
        Console.WriteLine("цикл {0}, выполнение внутри потока из пула {1}",
            i, Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(5000);
    }
}
```

ThreadPool.QueueUserWorkItem(20000);

Максимальное количество потоков: 1023

Потоков ввода-вывода доступно: 1000

цикл 0, выполнение внутри потока из пула 4

цикл 0, выполнение внутри потока из пула 3

цикл 0, выполнение внутри потока из пула 5

цикл 0, выполнение внутри потока из пула 6

цикл 0, выполнение внутри потока из пула 7

цикл 1, выполнение внутри потока из пула 3

цикл 1, выполнение внутри потока из пула 4

цикл 1, выполнение внутри потока из пула 5

цикл 1, выполнение внутри потока из пула 6

цикл 1, выполнение внутри потока из пула 7

цикл 2, выполнение внутри потока из пула 4

цикл 2, выполнение внутри потока из пула 3

цикл 2, выполнение внутри потока из пула 5

цикл 2, выполнение внутри потока из пула 6

цикл 2, выполнение внутри потока из пула 7

Пулы потоков очень просты в применении, однако обладают рядом ограничений:

- Все потоки в пуле потоков являются фоновыми - сделать поток из пула приоритетным не удастся
- Нельзя изменять приоритет или имя находящего в пуле потока
- Потоки в пуле подходят для выполнения только коротких задач (контрпример - проверка орфографии в Word)
- Блокирование потоков из пула может привести к снижению производительности

Пул потоков CLR

Компоненты, неявно использующие пул потоков:

- серверы приложений WCF, Remoting, ASP.NET и ASMX Web Services
- конструкции параллельного программирования
- класс BackgroundWorker (в C# 5.0 избыточный)
- асинхронные делегаты (в C# 5.0 избыточные)

Простейший способ явного запуска чего-либо в потоке – использование Task, до выхода .Net Framework 4.0 - ThreadPool.QueueUserWorkItem

```
// Task is in System.Threading.Tasks
Task.Run (() => Console.WriteLine ("Hello from the thread pool"));

ThreadPool.QueueUserWorkItem (notUsed => Console.WriteLine ("Hello"));
```



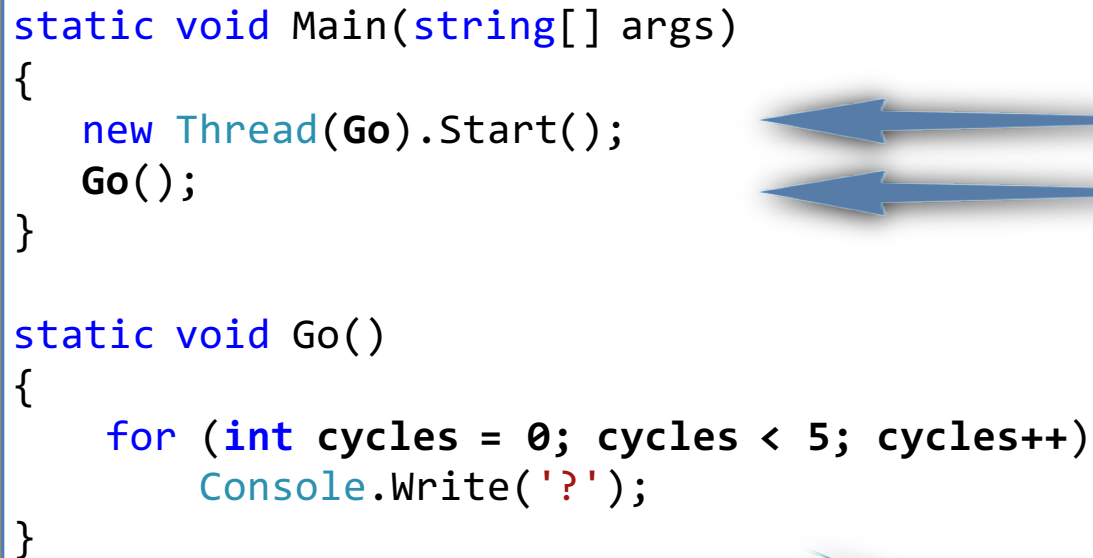
БАЗОВЫЕ СВЕДЕНИЯ О синхронизации

Обзор и ключевые понятия. Локальное и разделяемое состояние

CLR назначает каждому потоку свой стек, так что локальные переменные хранятся отдельно

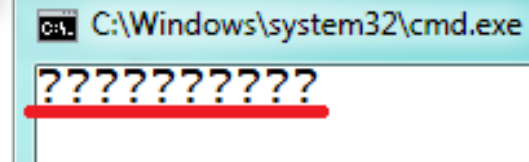
```
static void Main(string[] args)
{
    new Thread(Go).Start();
    Go();
}

static void Go()
{
    for (int cycles = 0; cycles < 5; cycles++)
        Console.Write('?');
}
```



Выполнить Go() в
новом потоке

Выполнить Go() в
главном потоке



Обзор и ключевые понятия. Создание потока

```
using System;
using System.Threading;

...

static void Main(string[] args)
{
    Thread t = new Thread(WriteY);
    t.Start();
    for (int i = 0; i < 1000; i++)
    {
        if (i % 100 == 0)
            Console.WriteLine();
        Console.Write("x");
    }
    Console.ReadKey();
}

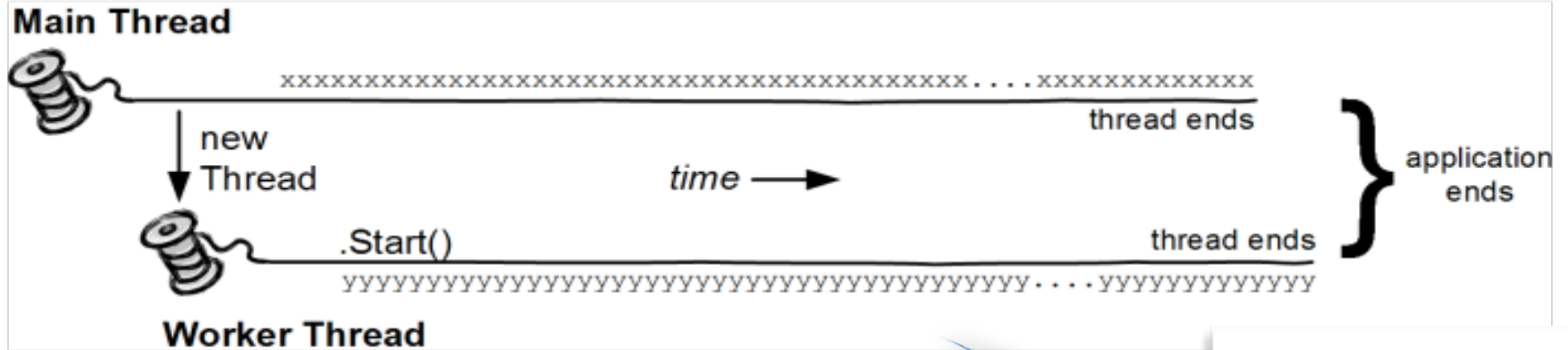
static void WriteY()
{
    for (int i = 0; i < 1000; i++)
    {
        if (i % 100 == 0)
            Console.WriteLine();
        Console.Write("y");
    }
}
```

```
public delegate void ThreadStart();
```

```

yyyyyyyyyyyyy
yyyyyyyyyyyyy
yyyyyy
xxxxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxxyyyyyy
yyyyyyyyyyyyy
yyyyyyyyyyyyy
yyyyyyyyyyyyy
yyyyyyyyyyyyy
yyyyyyyyyyyyy
yyyyyyyyyyyyy
yyyyyyyyyyyyy
```

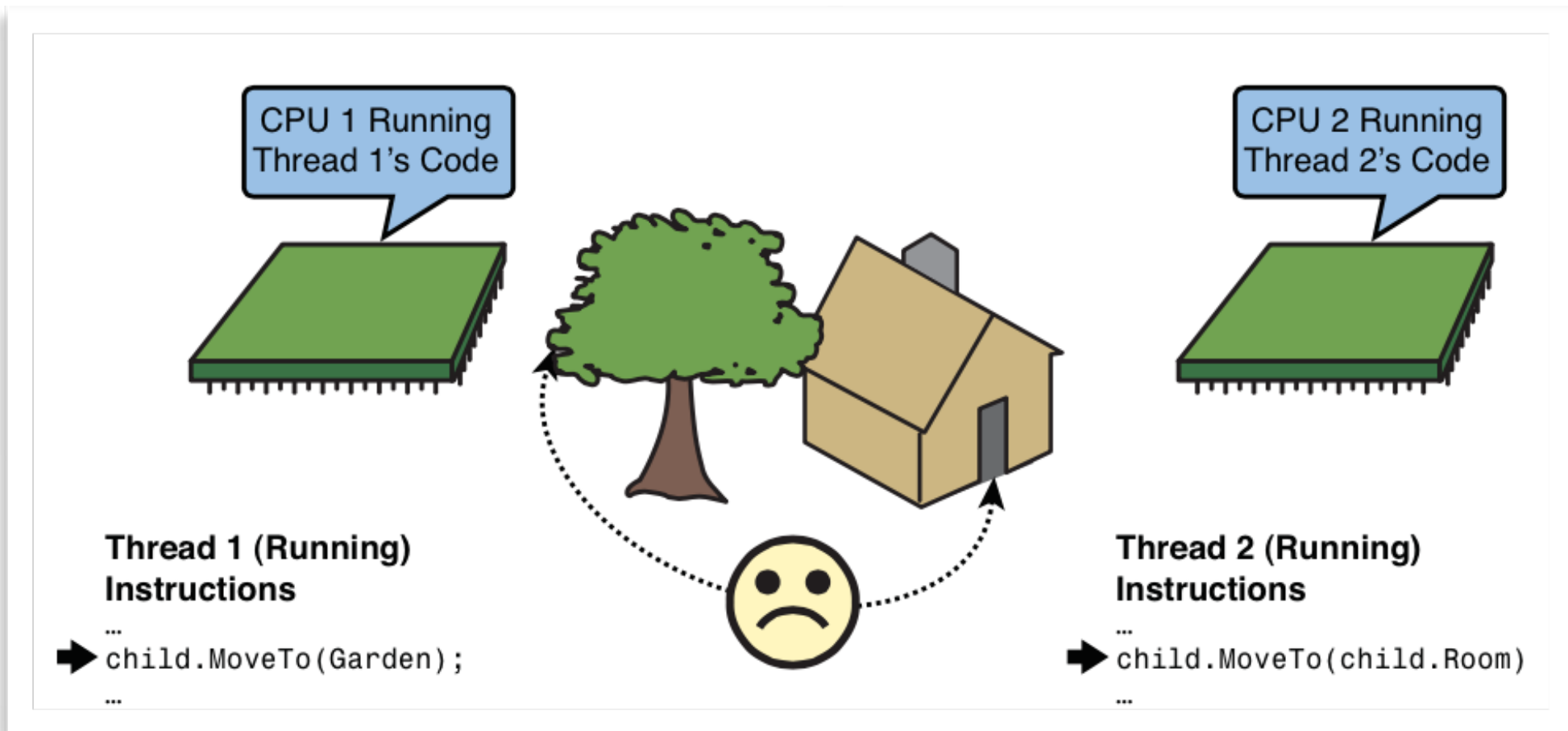

Обзор и ключевые понятия. Создание потока



Говорят, что поток **вытесняется**, когда его выполнение приостанавливается из-за внешних факторов типа квантования времени. В большинстве случаев поток не может контролировать, когда и где он будет вытеснен

This diagram illustrates thread preemption. It shows a vertical sequence of characters. The top section consists of ten lines of 'x' characters. The bottom section consists of ten lines of 'y' characters. A vertical line on the left side of the 'y' section indicates the start of the Worker Thread's execution. The 'y' characters are shown being interleaved with the 'x' characters, demonstrating how the Worker Thread's execution is preempted by the Main Thread's execution.

Обзор и ключевые понятия. Локальное и разделяемое состояние



Обзор и ключевые понятия . Локальное и разделяемое состояние

Потоки разделяют данные, относящиеся к тому же экземпляру объекта, что и сами потоки

```
class ThreadTest
{
    private bool done;
    static void Main(string[] args)
    {
        ThreadTest tt = new ThreadTest(); //общий экземпляр
        new Thread(tt.Go).Start();
        tt.Go();
    }

    private void Go()
    {
        if (!done)
        {
            done = true;
            Console.WriteLine("Done");
        }
    }
}
```



Обзор и ключевые понятия. Локальное и разделяемое состояние

Локальные переменные, захваченные лямбда-выражением или анонимным методом, могут быть также разделяемыми

```
class ThreadTest
{
    static void Main()
    {
        bool done = false;
        ThreadStart action = () =>
        {
            if (!done)
            {
                done = true;
                Console.WriteLine ("Done");
            }
        };
        new Thread (action).Start();
        action();
    }
}
```



Done

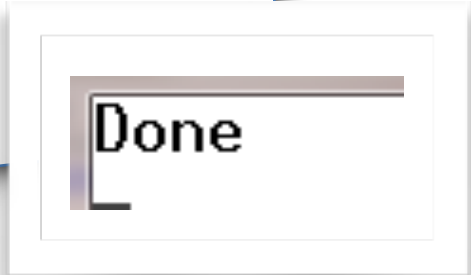
Обзор и ключевые понятия . Локальное и разделяемое состояние

Для статических полей работает другой способ разделения данных между потоками

```
class ThreadTest
{
    private static bool done;
    static void Main(string[] args)
    {
        ThreadTest tt = new ThreadTest();
        new Thread(tt.Go).Start();
        tt.Go();
    }

    private void Go()
    {
        if (!done)
        {
            done = true;
            Console.WriteLine("Done");
        }
    }
}
```

Фактически результат исполнения программы не определен: возможно (хотя и маловероятно), "Done" будет напечатано дважды



Done

Обзор и ключевые понятия. Локальное и разделяемое состояние

```
class ThreadTest
{
    private static bool done;
    static void Main(string[] args)
    {
        ThreadTest tt = new ThreadTest();
        new Thread(tt.Go).Start();
        tt.Go();
    }

    private void Go()
    {
        if (!done)
        {
            Console.WriteLine("Done");
            done = true;
        }
    }
}
```

Отсутствие потоковой безопасности

При изменении порядка вызовов в методе **Go()**, шансы увидеть "Done" напечатанным два раза повышаются радикально!



Done
Done

Разделяемое записываемое состояние может привести к разновидности не систематических ошибок, характерных для многопоточности

Обзор и ключевые понятия. Блокировка и безопасность потоков

Получение **эксклюзивной блокировки** (exclusive lock) на время чтения и записи разделяемых полей в C# обеспечивается при помощи оператора **lock**

```
class ThreadSafe
{
    static bool done;
    static object locker = new object();
    static void Main()
    {
        new Thread(Go).Start();
        Go();
    }
    static void Go()
    {
        lock (locker)
        {
            if (!done)
            {
                Console.WriteLine("Done");
                done = true;
            }
        }
    }
}
```

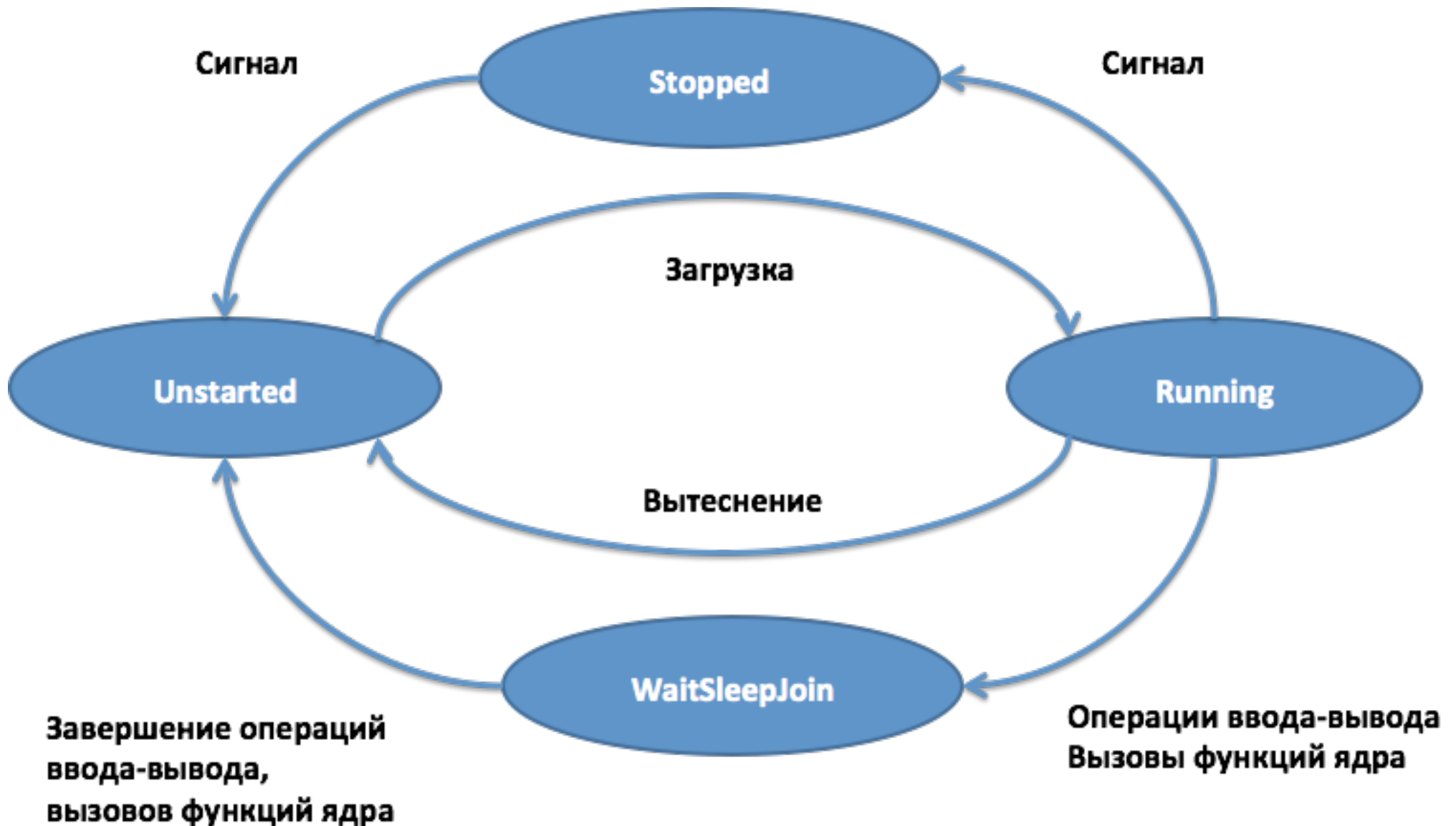
Когда два потока одновременно соперничают за блокировку (в нашем случае объекта **locker**), один поток переходит к ожиданию (блокируется), пока блокировка не освобождается. В данном случае это гарантирует, что только один поток может одновременно исполнять критическую секцию кода

Код, защищенный от неопределенности в многопоточном контексте, называется потокобезопасным!



Done

Обзор и ключевые понятия. Блокировка и безопасность потоков



Обзор и ключевые понятия. Блокировка и безопасность потоков

Временная приостановка (блокирование) – основной способ координации, или **синхронизации** действий потоков

Синхронизация потоков – это координирование действий потоков для получения предсказуемого результата в многопоточном контексте

Ожидание эксклюзивной блокировки – это одна из причин, по которым поток может блокироваться. Другая причина – если поток приостанавливается (Sleep) на заданный промежуток времени

```
Thread.Sleep(TimeSpan.FromHours(1)); // sleep for 1 hour
Thread.Sleep(500);                    // sleep for 500 milliseconds
```

Также поток может ожидать завершения другого потока, вызывая его метод **Join**

```
Thread t = new Thread(Go);
t.Start();
t.Join(); )); // Будучи заблокированным, поток не потребляет ресурсов CPU!
```

Важнейшие средства синхронизации

Синхронизация – это координирование параллельно выполняющихся действий с целью получения предсказуемых результатов

Средства синхронизации	Классы
Блокировка	Join, Sleep, SpinWait
Взаимно-исключающий доступ	lock, Monitor, Mutex, SpinLock
Сигнальные сообщения	AutoResetEvent, ManualResetEvent, ManualResetEventSlim
Семафоры	Semaphore, SemaphoreSlim
Атомарные операторы	Interlocked
Конкурентные коллекции	ConcurrentBag, ConcurrentQueue, ConcurrentDictionary, ConcurrentStack, BlockedCollection
Блокировки чтения-записи	ReaderWriterLock, ReaderWriterLockSlim
Шаблоны синхронизации	Barrier, CountdownEvent

Простейшие методы блокировки

Конструкция	Назначение
Sleep	Блокировка на указанное время
Join	Ожидание окончания другого потока
SpinWait	Вынуждает поток выполнять ожидание столько раз, сколько определено параметром <i>iterations</i>

Блокировочные конструкции

Конструкция	Назначение	Доступна из других процессов?	Скорость
lock	Гарантирует, что только один поток может получить доступ к ресурсу или секции кода	нет	быстро
Mutex	Гарантирует, что только один поток может получить доступ к ресурсу или секции кода.	да	средне
Semaphore	Гарантирует, что не более заданного числа потоков может получить доступ к ресурсу или секции кода.	да	средне

Сигнальные конструкции

Конструкция	Назначение	Доступна из других процессов?	Скорость
EventWaitHandle	Позволяет потоку ожидать сигнала от другого потока	да	средне
Wait and Pulse	Позволяет потоку ожидать, пока не выполнится заданное условие блокировки	нет	средне

Важнейшие средства синхронизации

Не блокирующие конструкции синхронизации

Конструкция	Назначение	Доступна из других процессов?	Скорость
Interlocked	Выполнение простых не блокирующих атомарных операций.	Да – через разделяемую память	очень быстро
volatile	Для безопасного не блокирующего доступа к полям.	Да – через разделяемую память	очень быстро

Важнейшие средства синхронизации. Блокировка

Когда поток остановлен в результате использования блокирующих конструкций, говорят, что он заблокирован. Будучи заблокированным, поток немедленно перестает получать время CPU, устанавливает свойство **ThreadState** в **WaitSleepJoin** и остается в таком состоянии, пока не разблокируется

Разблокировка может произойти в следующих четырех случаях (кнопка выключения питания не считается 😊):

- выполнится условие разблокировки
- истечет таймаут операции (если он был задан)
- по прерыванию через **Thread.Interrupt**
- по аварийному завершению через **Thread.Abort**

Поток не считается заблокированным, если его выполнение приостановлено не рекомендуемым методом **Suspend (Resume)**

Важнейшие средства синхронизации. Блокировка. **Sleeping** и **Spinning**

Вызов **Thread.Sleep** блокирует текущий поток на указанное время (либо до прерывания), по сути отпускает CPU и сообщает, что потоку не должно выделяться время в указанный период:

```
Thread.Sleep(0);    // отказаться от одного кванта времени CPU
Thread.Sleep(1000); // заснуть на 1000 миллисекунд
Thread.Sleep(TimeSpan.FromHours(1)); // заснуть на 1 час
Thread.Sleep(Timeout.Infinite);      // заснуть до прерывания
```

Класс **Thread** также предоставляет метод **SpinWait**, который не отказывается от времени CPU, а наоборот, загружает процессор в цикле на заданное количество итераций. 50 итераций эквивалентны паузе примерно в микросекунду (зависит от скорости и загрузки CPU)

Важнейшие средства синхронизации. Блокировка. Блокирование против ожидания в цикле

Поток может ожидать выполнения некоторого условия, непосредственно прокручивая цикл проверки

```
while (!proceed) ;  
...  
while (DateTime.Now < nextStartTime) ;
```

CLR и операционная система убеждены, что поток выполняет важные вычисления, ему выделяются соответствующие ресурсы

Поток, который крутится в таком состоянии, не считается заблокированным, в отличие от потока, ожидающего на **EventWaitHandle** (конструкции, обычно используемой для таких задач)

Важнейшие средства синхронизации. Блокировка. Ожидание завершения потока

Поток можно заблокировать до завершения другого потока вызовом метода **Join**

```
Thread t = new Thread(() => Console.ReadLine());  
t.Start();  
t.Join();    // ожидать, пока поток не завершится  
Console.WriteLine("Thread t's ReadLine complete!");
```

Метод **Join** может также принимать в качестве аргумента **timeout** - в миллисекундах или как **TimeSpan**. Если указанное время истекло, а поток не завершился, **Join** возвращает **false**. **Join** с **timeout** функционирует как **Sleep**

```
Thread.Sleep(1000);  
Thread.CurrentThread.Join(1000);
```

Одинаковый результат!

Блокирование и потоковая безопасность

Блокировка обеспечивает монопольный доступ и используется, чтобы обеспечить выполнение одной секции кода только одним потоком одновременно

```
class ThreadUnsafe
{
    static int value1, value2;
    public static void Go()
    {
        if (value2 != 0)
            Console.WriteLine(value1 / value2);
        value2 = 0;
    }
}
```

Если за блокировку борются несколько потоков, они ставятся в очередь ожидания – «ready queue» – и обслуживаются по принципу «первым пришел – первым обслужен»

Только один поток может одновременно заблокировать объект синхронизации (locker), а все другие конкурирующие потоки будут приостановлены, пока блокировка не будет снята

```
class ThreadSafe
{
    static object locker = new object();
    public static int value1, value2;
    static void Go()
    {
        lock (locker)
        {
            if (value2 != 0)
                Console.WriteLine(value1 / value2);
            value2 = 0;
        }
    }
}
```

ThreadState.WaitSleepJoin

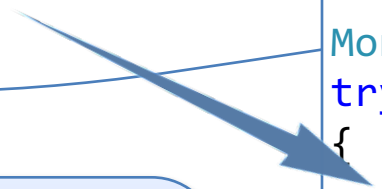
Блокирование и потоковая безопасность

Оператор **lock** языка C# фактически является синтаксическим сокращением для вызовов методов **Monitor.Enter** и **Monitor.Exit** в рамках блоков **try-finally**

```
lock (locker)
{
    if (value2 != 0)
        Console.WriteLine(value1 / value2);
    value2 = 0;
}
```

Вызов **Monitor.Exit** без предшествующего вызова **Monitor.Enter** для того же объекта синхронизации вызовет исключение

Класс **Monitor** также предоставляет метод **TryEnter**, позволяющий задать время ожидания в миллисекундах или в виде **TimeSpan**



```
Monitor.Enter(locker);
try
{
    if (value2 != 0)
        Console.WriteLine(value1 / value2);
    value2 = 0;
}
finally
{
    Monitor.Exit(locker);
}
```

Блокирование и потоковая безопасность

Если блокировка никогда не будет получена, вход в блок try-finally никогда не выполнится

Monitor - класс

Monitor - методы

Enter - метод

Enter - метод (Object)

Enter - метод (Object, Boolean)

Exit - метод

IsEntered - метод

Pulse - метод

PulseAll - метод

TryEnter - метод

TryEnter - метод (Object)

TryEnter - метод (Object, Boolean)

TryEnter - метод (Object, Int32)

TryEnter - метод (Object, TimeSpan)

TryEnter - метод (Object, Int32, Boolean)

TryEnter - метод (Object, TimeSpan, Boolean)

Wait - метод


```
bool lockTaken = false;
try
{
    Monitor.Enter(locker, ref lockTaken);
    // Do your stuff...
}
finally
{
    if (lockTaken) Monitor.Exit(locker);
}
```

Блокирование и потоковая безопасность. Выбор объекта синхронизации

Любой объект, видимый взаимодействующим потокам, может быть использован как объект синхронизации, если это объект **ссылочного** типа. Также строго рекомендуется, чтобы объект синхронизации был `private`-полем класса или статическим полем, во избежание случайного воздействия внешнего кода, блокирующего этот объект.

```
List <string> list = new List <string>();  
  
void Test()  
{  
    lock (list)  
    {  
        list.Add("Item 1");  
        . . .  
    }  
    . . .  
}
```

Объектом синхронизации
вполне может стать сам
защищаемый объект



Блокирование и потоковая безопасность. Выбор объекта синхронизации

Обычно используется выделенное поле, так как это позволяет точнее контролировать область видимости и степень детализации блокировки. Использование объекта или типа в качестве объекта синхронизации не одобряется, так как предполагает public-область видимости объекта синхронизации

```
lock (this)
{
    ...
}
```



```
lock (typeof(Widget)) // Для защиты статических данных
{
    ...
}
```



Логика блокирования не инкапсулирована - труднее предотвратить взаимоблокировки и избыточные блокировки

Вложенные блокировки

Поток может неоднократно блокировать один и тот же объект многократными вызовами **Monitor.Enter** или вложенными **lock**-ами. Объект будет освобожден, когда будет выполнено соответствующее количество раз **Monitor.Exit** или произойдет выход из самой внешней конструкции **lock**

```
static object x = new object();
static void Main()
{
    lock (x)
    {
        Console.WriteLine("I have the lock");
        Nest();
        Console.WriteLine ("I still have the lock");
    } // Здесь блокировка будет снята!
}
static void Nest()
{
    lock (x)
    { ... }
    //Блокировка еще не снята!
}
```


Когда блокировать

Как правило, любое записываемое поле, доступное нескольким потокам, должно читаться и записываться с блокировкой. Даже в самом простом случае, операции присваивания одиночному полю, необходима синхронизация

```
class ThreadUnsafe
{
    static int x;
    static void Increment() { x++; }
    static void Assign() { x = 123; }
}
```

Ни инкремент, ни
присваивание не
являются
 потокобезопасными

```
class ThreadSafe
{
    static object locker = new object();
    static int x;
    static void Increment()
    {
        lock (locker) x++;
    }
    static void Assign()
    {
        lock (locker) x = 123;
    }
}
```

Когда блокировать

Операции а-ля инкрементирования (а иногда и чтения/записи) не являются атомарными (не выполняются неделимым образом на лежащем в основе процессоре)

Компилятор, среда CLR и процессор имеют право изменять порядок следования инструкций и кэшировать переменные в регистрах центрального процессора с целью улучшения производительности до тех пор, пока это не изменяет поведение однопоточной программы (или многопоточной, использующей блокировки). Совокупность правил перестановок таких инструкций называется моделью памяти

Модель памяти .NET

Тип перестановки	Перестановка разрешена
Загрузка-загрузка	Да
Загрузка-запись	Да
Запись-загрузка	Да
Запись-запись	Нет

Неблокирующая синхронизация

Необходимость синхронизации в случаях присвоения значения или увеличения значения поля

Эксклюзивная блокировка чревата соответствующими накладными расходами

Конструкции неблокирующей синхронизации .NET Framework позволяют выполнить простые операции без блокирования, приостановок и ожидания

Используются атомарные операции, а также чтение и запись с семантикой «volatile»

Неблокирующая синхронизация. Атомарность и Interlocked

Инструкция является атомарной, если она выполняется как единая, неделимая команда. Строгая атомарность препятствует любой попытке вытеснения. Простое чтение или присвоение значения полю в 32 бита или менее является атомарным. Операции с большими полями не атомарны, так как являются комбинацией более чем одной операции чтения/записи

```
class Atomicity
{
    static int x;
    static int y;
    static long z;

    static void Test()
    {
        long myLocal;
        x = 3;           // Атомарная операция
        z = 3;           // Не атомарная (z – 64-битная переменная)
        myLocal = z;     // Не атомарная (z is 64 bits)
        y += x;          // Не атомарная (операции чтения и записи)
        x++;             // Не атомарная (операции чтения и записи)
    }
}
```

Неблокирующая синхронизация. Атомарность и Interlocked

Метод	Назначение
CompareExchange	Безопасно проверяет два значения на эквивалентность. Если они эквивалентны, изменяет одно из значений на третье
Decrement	Безопасно уменьшает значение на 1
Exchange	Безопасно меняет два значения местами
Increment	Безопасно увеличивает значение на 1

Неблокирующая синхронизация. Атомарность и Interlocked

```
class ThreadUnsafe
{
    public static int x = 100;

    public static void Go()
    {
        for (int i = 0; i < 100; i++)
            x--;
    }
}
```

Неблокирующая синхронизация. Атомарность и Interlocked

```
static long sum;

static void Main()                                // sum
{
    // Простой increment/decrement:
    Interlocked.Increment(ref sum);                // 1
    Interlocked.Decrement(ref sum);                // 0
    // Сложение/вычитание:
    Interlocked.Add(ref sum, 3);                    // 3
    // Чтение 64-битного поля:
    Console.WriteLine(Interlocked.Read(ref sum));  // 3
    // Запись 64-битного поля после чтения предыдущего значения:
    Console.WriteLine(Interlocked.Exchange(ref sum, 10)); // 10
    // Обновление поля только если оно соответствует
    // определенному значению(10):сравнение первого аргумента с третьим и
    // если они равны, заменяет первый аргумент на второй
    Interlocked.CompareExchange(ref sum, 123, 10); // 123
}
```

Использование Interlocked более эффективно, чем lock, так как при этом в принципе отсутствует блокировка – и соответствующие накладные расходы на временную приостановку потока

Сигнализирующие конструкции. Методы Wait, Pulse и PulseAll

Поток Т выполняется в кодовом блоке lock, и ему требуется доступ к ресурсу R, который временно недоступен. Что же тогда делать потоку Т? Если поток Т войдет в организованный в той или иной форме цикл опроса, ожидая освобождения ресурса R, то тем самым он свяжет соответствующий объект, блокируя доступ к нему других потоков

Сообщение между потоками организуется в С# с помощью методов Wait(), Pulse() и PulseAll() (класс Monitor)

Пример
TickTock

Interrupt и Abort

Заблокированный поток может быть преждевременно разблокирован двумя путями:

- с помощью **Thread.Interrupt**
- с помощью **Thread.Abort**

Вызов **Interrupt** для заблокированного потока принудительно освобождает его с генерацией исключения **ThreadInterruptedException**

```
Thread t = new Thread(delegate()  
{  
    try  
    {  
        Thread.Sleep(Timeout.Infinite);  
    }  
    catch (ThreadInterruptedException)  
    {  
        Console.WriteLine("Forcibly ");  
    }  
    Console.WriteLine("Woken!");  
});
```

```
t.Start();  
t.Interrupt();
```

Forcibly Woken!

Interrupt и Abort

Вызов **Abort** для заблокированного потока принудительно освобождает его с генерацией исключения **ThreadAbortException**. Кроме того, это исключение будет повторно сгенерировано в конце блока **catch** (в попытке успокоить поток навеки), если только в блоке **catch** не будет вызван **Thread.ResetAbort**. До вызова **Thread.ResetAbort** **ThreadState** будет иметь значение **AbortRequested**

Отличие между **Interrupt** и **Abort** состоит в том, что происходит, если их вызвать для неблокированного потока. Если **Interrupt** ничего не делает, пока поток не дойдет до следующей блокировки, то **Abort** генерирует исключение непосредственно в том месте, где сейчас находится поток – может быть, даже не в вашем коде. Аварийное завершение неблокированного потока может иметь существенные последствия

Interrupt и Abort

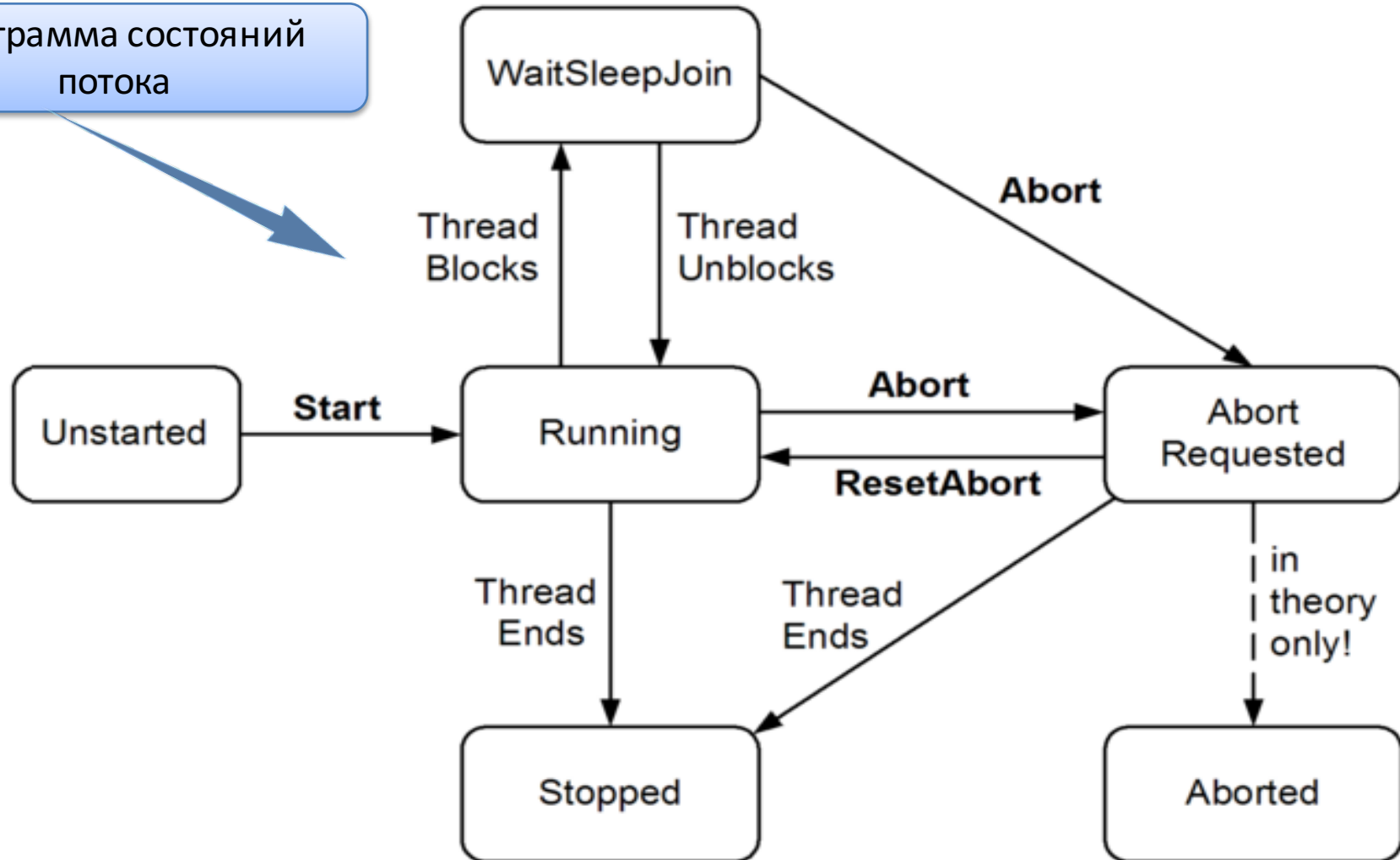
```
public static void Main()
{
    // создаем и запускаем поток
    var th = new Thread(ThreadProc);
    th.Start(); // ждем 3 секунды
    Thread.Sleep(3000); // пытаемся прервать работу потока th
    th.Abort(); // ждем завершения потока
    th.Join(); // ... но не дождемся, так как поток сам себя "воскресил"
}

public static void ThreadProc()
{
    while (true)
    {
        try
        {
            Console.WriteLine("Do some work...");
            Thread.Sleep(1000);
        }
        catch (ThreadAbortException e)
        {
            // отлавливаем попытку уничтожения и отменяем её
            Console.WriteLine("Somebody tries to kill me!");
            Thread.ResetAbort();
        }
    }
}
```

Состояния потока

ThreadState это комбинация трех уровней состояний с использованием битовых флагов, члены перечисления в пределах каждого уровня являются взаимоисключающими

Диаграмма состояний
потока



Состояния потока

Имя члена	Описание
Running	Поток был запущен, он не заблокирован, и нет ожидающего исключения <code>ThreadAbortException</code>
StopRequested	Поток получает запрос на остановку. Предназначено только для внутреннего использования
SuspendRequested	Запрашивается приостановка работы потока
Background	Поток выполняется как фоновый поток, в противоположность потокам переднего плана. Это состояние управляется заданием свойства <code>Thread.IsBackground</code>
Unstarted	Метод <code>Thread.Start</code> не был вызван для потока
Stopped	Поток был остановлен
WaitSleepJoin	Поток заблокирован. Это может произойти в результате вызова метода <code>Thread.Sleep</code> или метода <code>Thread.Join</code> , в результате запроса блокировки, например при вызове метода <code>Monitor.Enter</code> или <code>Monitor.Wait</code> или в результате ожидания объекта синхронизации потока, такого как <code>ManualResetEvent</code>
Suspended	Поток был приостановлен
AbortRequested	Метод <code>Thread.Abort</code> был вызван для потока, но поток еще не получил исключение <code>System.Threading.ThreadAbortException</code> , которое попытается завершить его
Aborted	Состояние потока включает в себя значение <code>AbortRequested</code> , и поток теперь не выполняет работу, но его состояние еще не изменилось на <code>Stopped</code>

Состояния потока

Следующий код относит ThreadState к одному из четырех наиболее используемых значений: Unstarted, Running, WaitSleepJoin и Stopped:

```
public static ThreadState SimpleThreadState(ThreadState ts)
{   return ts & (ThreadState.Unstarted |
                ThreadState.WaitSleepJoin |
                ThreadState.Stopped); }
```

Структура SpinLock

В .NET 4 появилась структура `SpinLock`, которая может применяться в случае, когда накладные расходы, связанные с объектами блокировки (`Monitor`), оказываются слишком высокими из-за сборки мусора

Эта структура особенно полезна при большом количестве блокировок (например, для каждого узла в списке) и чрезмерно коротких периодах их удержания

За исключением различий в архитектуре, структура `SpinLock` в плане применения очень похожа на класс `Monitor` - получение блокировки осуществляется с помощью методов **`Enter`** или **`TryEnter`**, а снятие — с помощью метода **`Exit`**

При передаче экземпляров `SpinLock` следует соблюдать осторожность. Из-за того, что `SpinLock` определена как `struct`, ее присваивание приводит к созданию копии. Поэтому экземпляры `SpinLock` должны всегда передаваться по ссылке.

Потоковая безопасность

Потокобезопасный код – это код, не имеющий никаких неопределенностей при любых сценариях многопоточного исполнения

Средства достижения потокобезопасности - блокировки и сокращение возможностей взаимодействия между потоками

- Помещение больших разделов кода или даже доступ ко всему объекту внутрь монопольной блокировки
- Минимизация взаимодействия потоков за счет сведения к минимуму их разделяемых данных
- Использование режима автоматического блокирования

Типы общего назначения редко являются полностью потокобезопасными по следующим причинам:

- разработка с учетом полной потоковой безопасности может быть очень трудоемкой, особенно если тип имеет много полей (каждое поле потенциально может участвовать в многопоточном взаимодействии)
- потоковая безопасность может сказаться на производительности (независимо от того, используется ли реально многопоточность)
- потокобезопасный тип не обязательно делает использующую его программу потокобезопасной, а дальнейшая работа по ее обеспечению может сделать потокобезопасность типа избыточной

Потокобезопасность и типы .NET Framework

Почти все непримитивные типы .NET Framework непотокобезопасны, и все же они могут использоваться в многопоточном коде, если любой доступ к любому объекту защищен блокировкой

```
class ThreadSafe
{
    static List <string> list = new List <string>();
    static void Main()
    {
        new Thread(AddItems).Start();
        new Thread(AddItems).Start();
    }

    static void AddItems()
    {
        for (int i = 0; i < 100; i++)
            lock (list)
                list.Add("Item " + list.Count);
        string[] items;
        lock (list)
            items = list.ToArray();
        foreach (string s in items)
            Console.WriteLine(s);
    }
}
```

```
if(!list.Contains(newItem))
    list.Add(newItem);
```

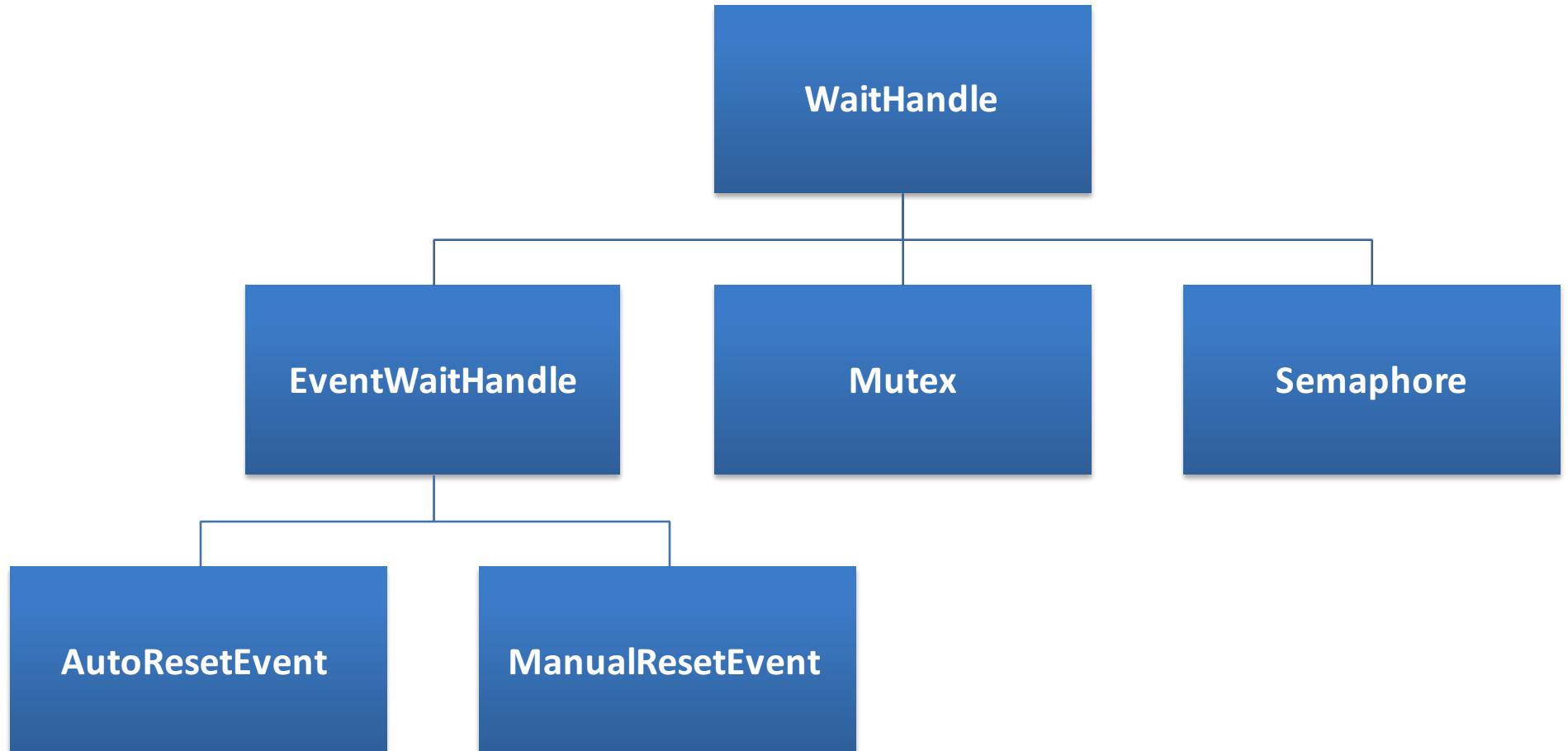
Дескрипторы ожидания (Wait Handles)

Оператор **lock** (**Monitor.Enter/Monitor.Exit**) является самым подходящим средством для организации монопольного доступа к ресурсу или секции кода, но есть задачи синхронизации типа подачи сигнала начала работы ожидающему потоку

Конструкции синхронизации Win32 API, доступные в .NET Framework в виде классов **EventWaitHandle**, **Mutex** и **Semaphore**. Некоторые из них практичнее других: **Mutex**, например, по большей части дублирует возможности **lock**, в то время как **EventWaitHandle** предоставляет уникальные возможности сигнализации

Все три класса основаны на абстрактном классе **WaitHandle**, но весьма отличаются по поведению. Одна из общих особенностей – это способность именования, делающая возможной работу с потоками не только одного, но и разных процессов.

Потребность в синхронизации на основе подачи сигналов возникает, когда один поток ждет прихода уведомления от другого потока



Класс Mutex

Класс Mutex (mutual exclusion — взаимное исключение или мьютекс) позволяет обеспечить синхронизацию среди множества процессов (**lock** быстрее в сотни раз)

Допускает наличие только одного владельца - только один поток может получить блокировку и иметь доступ к защищаемым мьютексом синхронизированным областям кода

Метод **WaitOne** для **Mutex** получает исключительную блокировку, блокируя поток, если это необходимо. Исключительная блокировка может быть снята вызовом метода **ReleaseMutex**. Аналогично оператору **lock** в C#, **Mutex** может быть освобожден только из того же потока, что его захватил

Если приложение завершается без вызова **ReleaseMutex**, CLR освобождает мьютекс автоматически

Пример. Два простых приложения - первое будет записывать случайные числа в файл, второе считывать их из этого файла и выводить на экран. Третье приложение, будет запускать первые два

Класс Semaphore (SemaphoreSlim)

Семафор предоставляет одновременный доступ к общему ресурсу не одному, а нескольким потокам. Поэтому семафор используется для синхронизации целого ряда ресурсов

Семафор управляет доступом к общему ресурсу, используя счетчик. Поток, которому требуется доступ к общему ресурсу, пытается получить разрешение от семафора. Если значение счетчика семафора больше нуля, то поток получает разрешение, а счетчик семафора декрементируется. В противном случае поток блокируется до тех пор, пока не получит разрешение. Когда же потоку больше не требуется доступ к общему ресурсу, он высвобождает разрешение, а счетчик семафора инкрементируется. Если разрешения ожидает другой поток, то он получает его в этот момент. Количество одновременно разрешаемых доступов указывается при создании семафора. Так, если создать семафор, одновременно разрешающий только один доступ, то такой семафор будет действовать как мьютекс

пример

Декларативная блокировка

Вместо ручной блокировки можно осуществлять блокировки декларативно, используя наследование от `ContextBoundObject` и применяя атрибут `Synchronization`, фактически поручая CLR делать блокировки автоматически

```
[Synchronization]
public class AutoLock : ContextBoundObject
{
    public void Demo()
    {
        Console.Write("Старт...");
        Thread.Sleep(5000);           // Поток не может быть вытеснен здесь
        Console.WriteLine("стоп");    // спасибо автоматической блокировке!
    }
}

public class Test
{
    public static void Main()
    {
        AutoLock safeInstance = new AutoLock();
        new Thread(safeInstance.Demo).Start(); // Запустить метод
        new Thread(safeInstance.Demo).Start(); // Демо 3 раза
        safeInstance.Demo();                   // одновременно.
    }
}
```

▲ **WaitHandle** - класс

WaitHandle - конструктор

▸ WaitHandle - поля

▲ WaitHandle - методы

Close - метод

▸ Dispose - метод

IDisposable.Dispose - метод

MemberwiseClone - метод

▸ SignalAndWait - метод

▸ WaitAll - метод

▸ WaitAny - метод

▸ WaitOne - метод

▲ WaitHandle - свойства

Handle - свойство

SafeWaitHandle - свойство

Сигнализирование с помощью дескрипторов ожидания событий (Event wait handle)

Дескрипторы ожидания событий являются простейшими из сигнальных конструкций в C#

Для использования системных событий из управляемого кода - классы `ManualResetEvent`, `AutoResetEvent`, `ManualResetEventSlim` (.NET 4) и `CountdownEvent` (.NET 4) (`System.Threading`)

Эти классы применяются в тех случаях, когда один поток ожидает появления некоторого события в другом потоке. Как только такое событие появляется, второй поток уведомляет о нем первый поток, позволяя тем самым возобновить его выполнение

AutoResetEvent – наиболее часто используемый **WaitHandle**-класс и основная конструкция синхронизации, наряду с **lock**

Сигнализирующие конструкции

Конструкция	Назначение	Cross-process?	Overhead
AutoResetEvent	Разблокирует поток как только он получит сигнал от другого потока	Да	1000ns
ManualResetEvent	Позволяет разблокировать поток на неопределенное время, когда он получает сигнал от друга (до сброса)	Да	1000ns
ManualResetEventSlim (Framework 4.0)		-	40ns
CountdownEvent (Framework 4.0)	Позволяет разблокировать поток, когда он получает определенное количество сигналов от другого потока	-	40ns
Barrier (Framework 4.0)	Реализует барьер выполнения потоков	-	80ns
Wait and Pulse	Позволяет блокировать поток до выполнения пользовательского условия	-	120ns for a Pulse

Сигнализирование с помощью дескрипторов ожидания событий. EventWaitHandle

Конструктор `EventWaitHandle` также позволяет создавать именованные `EventWaitHandle`, способные действовать через границы процессов. Имя задается обыкновенной строкой и может быть любым (если задаваемое имя уже используется на компьютере, возвращается ссылка на существующий `EventWaitHandle`, в противном случае операционная система создает новый)

```
EventWaitHandle wh = new EventWaitHandle(false,  
    EventResetMode.AutoReset, "MyCompany.MyApp.SomeName");
```

Сигнализирование с помощью дескрипторов ожидания событий. `AutoResetEvent`

AutoResetEvent очень похож на турникет – один билет позволяет пройти одному человеку. Приставка «auto» в названии относится к тому факту, что открытый турникет автоматически закрывается или «сбрасывается» после того, как позволяет кому-нибудь пройти. Поток блокируется у турникета вызовом **WaitOne** (ждать (*wait*) у данного (*one*) турникета, пока он не откроется), а билет вставляется вызовом метода **Set**. Если несколько потоков вызывают **WaitOne**, за турникетом образуется очередь. Билет может «вставить» любой поток – другими словами, любой (неблокированный) поток, имеющий доступ к объекту **AutoResetEvent**, может вызвать **Set**, чтобы пропустить один заблокированный поток

Если **Set** вызывается, когда нет ожидающих потоков, хэндл будет находиться в открытом состоянии, пока какой-нибудь поток не вызовет **WaitOne**. Эта особенность помогает избежать гонок между потоком, подходящим к турникету, и потоком, вставляющим билет («опа, билет вставлен на микросекунду раньше, очень жаль, но вам придется подождать еще сколько-нибудь!»). Однако многократный вызов **Set** для свободного турникета не разрешает пропустить за раз целую толпу – сможет пройти только один, все остальные билеты будут потрачены впустую

Сигнализирование с помощью дескрипторов ожидания событий. **AutoResetEvent**

AutoResetEvent позволяет потокам взаимодействовать друг с другом путем передачи сигналов. Как правило, этот класс используется, когда потокам требуется исключительный доступ к ресурсу. Поток ожидает сигнала, вызывая метод `WaitOne` при возникновении **AutoResetEvent** (сигнальное состояние вызов `Set`). Если **AutoResetEvent** находится в несигнальном состоянии, поток будет заблокирован, ожидая сигнала потока, в настоящий момент контролирующего ресурс, о том, что ресурс доступен (для этого вызывается метод `Set`).

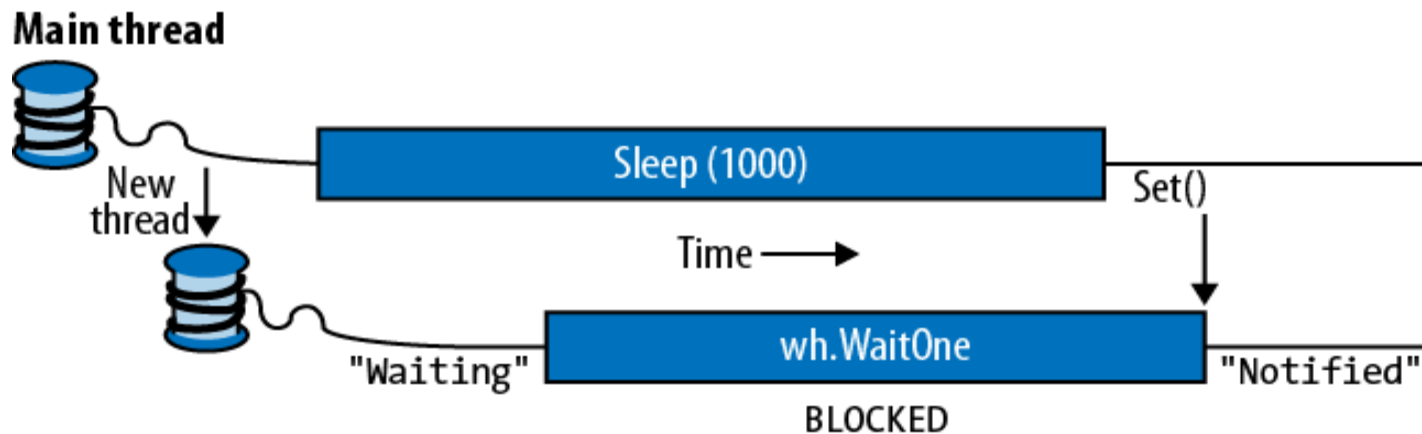
Если поток вызывает метод `WaitOne`, а **AutoResetEvent** находится в сигнальном состоянии, поток не блокируется. **AutoResetEvent** немедленно освобождает поток и возвращается в несигнальное состояние.

Вызов `Set` сигнализирует событию **AutoResetEvent** о необходимости освобождения ожидающего потока. Событие **AutoResetEvent** остается в сигнальном состоянии до освобождения одного ожидающего потока, а затем возвращается в несигнальное состояние. Если нет ожидающих потоков, состояние остается сигнальным бесконечно.

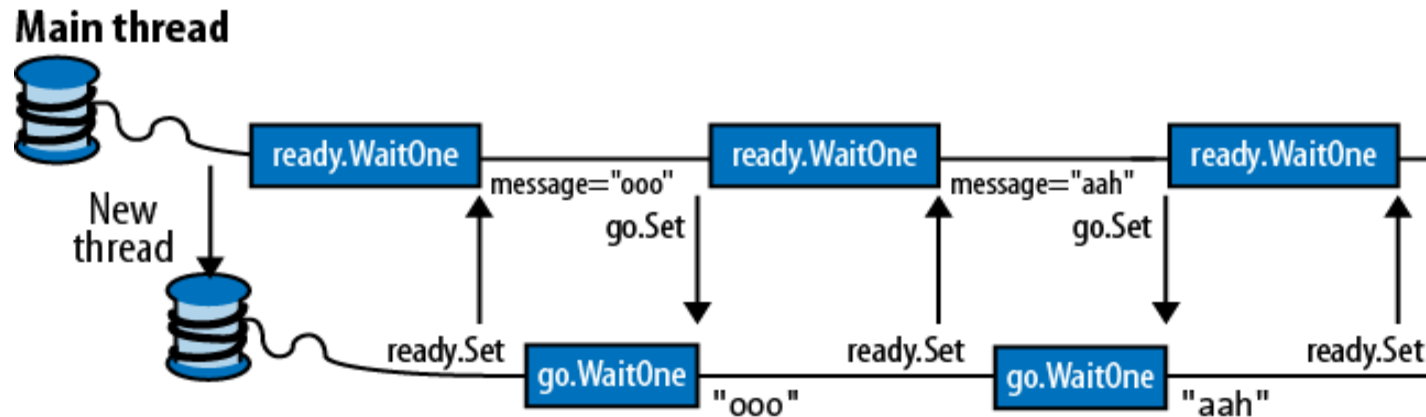
Сигнализирование с помощью дескрипторов ожидания событий. AutoResetEvent

```
class BasicWaitHandle
{
    static EventWaitHandle waitHandle = new AutoResetEvent(false);
    //EventWaitHandle wh = new EventWaitHandle(false, EventResetMode.AutoReset);
    static void Main()
    {
        new Thread(Waiter).Start();
        Thread.Sleep(1000);           // Pause for a second...
        waitHandle.Set();             // Wake up the Waiter.
    }

    static void Waiter()
    {
        Console.WriteLine("Waiting...");
        waitHandle.WaitOne();         // Wait for notification
        Console.WriteLine("Notified");
    }
}
```



Сигнализирование с помощью дескрипторов ожидания событий. `AutoResetEvent`. Шаблон «Получите и распишитесь» («Get and sign»)



Предположим, мы хотим исполнять задачи в фоновом режиме без затрат на создание каждый раз нового потока для новой задачи. Этой цели можно достигнуть, используя единственный рабочий поток с постоянным циклом, ожидающим появление задачи. Получив задачу, он приступает к ее выполнению. После окончания выполнения поток снова переходит в режим ожидания

Сигнализирование с помощью дескрипторов ожидания событий. AutoResetEvent

```
static EventWaitHandle _ready = new AutoResetEvent(false);
static EventWaitHandle _go = new AutoResetEvent(false);
static readonly object _locker = new object();
static string _message;

static void Main()
{
    new Thread(Work).Start();
    _ready.WaitOne(); // First wait until worker is ready
    lock (_locker) _message = "ooo"; // Tell worker to go
    _go.Set();

    _ready.WaitOne();
    lock (_locker) _message = "ahhh"; // Give the worker another message
    _go.Set();
    _ready.WaitOne();
    lock (_locker)
    {
        _go.Set();
    }
}

static void Work()
{
    while (true)
    {
        _ready.Set(); // Сообщаем о готовности
        _go.WaitOne(); // Ожидаем сигнала начать...
        lock (_locker)
        {
            if (_message == null) return; // Gracefully exit
            Console.WriteLine(_message);
        }
    }
}
```


Сигнализирование с помощью дескрипторов ожидания событий. AutoResetEvent. Очередь Поставщик/Потребитель (Producer/Consumer)

Поставщик ставит задачи в очередь, Потребитель извлекает задачи из очереди в рабочем потоке, при этом вызывающий поток не блокируется, если рабочий все еще занят предыдущей задачей

Потребителей может быть несколько, каждый обслуживает одну и ту же очередь, но в своем потоке

Пример

Сигнализирование с помощью дескрипторов ожидания событий. ManualResetEvent

ManualResetEvent – не сбрасывается автоматически, после того как поток проходит через WaitOne, и действует как флажок – Set открывает его, позволяя пройти любому количеству потоков, вызвавших WaitOne. Reset закрывает флажок, потенциально накапливая очередь ожидающих следующего открытия

```
var signal = new ManualResetEvent (false);

new Thread (() =>
{
    Console.WriteLine ("Waiting for signal...");
    signal.WaitOne();
    signal.Dispose();
    Console.WriteLine ("Got signal!");
}).Start();

Thread.Sleep(2000);
signal.Set();           // "Open" the signal
```

Сигнализирование с помощью дескрипторов ожидания событий. CountdownEvent

Класс CountdownEvent позволяет ожидать на более чем одном потоке (.NET Framework 4.0)

```
static CountdownEvent _countdown = new CountdownEvent (3);

static void Main()
{
    new Thread (SaySomething).Start ("I am thread 1");
    new Thread (SaySomething).Start ("I am thread 2");
    new Thread (SaySomething).Start ("I am thread 3");
    _countdown.Wait();    // Blocks until Signal has been called 3 times
    Console.WriteLine ("All threads have finished speaking!");
}

static void SaySomething (object thing)
{
    Thread.Sleep (1000);
    Console.WriteLine (thing);
    _countdown.Signal();
}
```

I am thread 3
I am thread 2
I am thread 1
All threads have finished speaking!

WaitAny, WaitAll и SignalAndWait

Методы WaitAny, WaitAll и SignalAndWait выполняют атомарные операции сигнализации и ожидания на множестве дескрипторов, которые могут быть отличающихся типов (включая Mutex и Semaphore)

SignalAndWait в рамках единой атомарной операции вызывает WaitOne для одного WaitHandle, и Set – для другого.

Классическим вариантом использования этого метода является использование с парой EventWaitHandle для подготовки встречи двух потоков в нужной точке в нужное время

```
WaitHandle.SignalAndWait(wh1, wh2);  
WaitHandle.SignalAndWait(wh2, wh1);
```

WaitHandle.WaitAny ожидает освобождения одного (любого) WaitHandle из переданного ему списка, WaitHandle.WaitAll ожидает освобождения сразу всех переданных ему WaitHandle. Используя аналогию с турникетом метро, эти методы организуют общую очередь одновременно для всех турникетов – с прохождением через первый открывшийся турникет (WaitAny) или с ожиданием, пока они не откроются все (WaitAll).

Класс Barrier

Класс Barrier – это сигнальная конструкция, которая появилась в .Net Framework 4.0. Он реализует барьер потока исполнения, который позволяет множеству потоков встречаться в определенном месте во времени (для сценариев, когда работа разбивается на множество задач и должна позже снова объединяться в одно целое). Этот класс очень быстрый и эффективный

Для использования класса Barrier необходимо:

- Создать экземпляр, указав количество потоков, которые будут встречаться одновременно (можно изменить это значение позднее путем вызова методов AddParticipants/RemoveParticipants)
- Каждый поток, ожидающий встречи, должен вызвать метод SignalAndWait

Класс Barrier

```
static Barrier _barrier = new Barrier (3);

static void Main()
{
    new Thread (Speak).Start();
    new Thread (Speak).Start();
    new Thread (Speak).Start();
}

static void Speak()
{
    for (int i = 0; i < 5; i++)
    {
        Console.Write (i + " ");
        _barrier.SignalAndWait();
    }
}
```

Output : 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4

Класс Barrier

Очень полезной возможностью класса Barrier является возможность указать при его конструировании *постдействие* (*post-phase action*). Этот делегат выполняется после того, как метод SignalAndWait вызван *n* раз, но *перед* тем, как потоки будут разблокированы

```
static Barrier _barrier = new Barrier (3, barrier => Console.WriteLine());
```

0 0 0

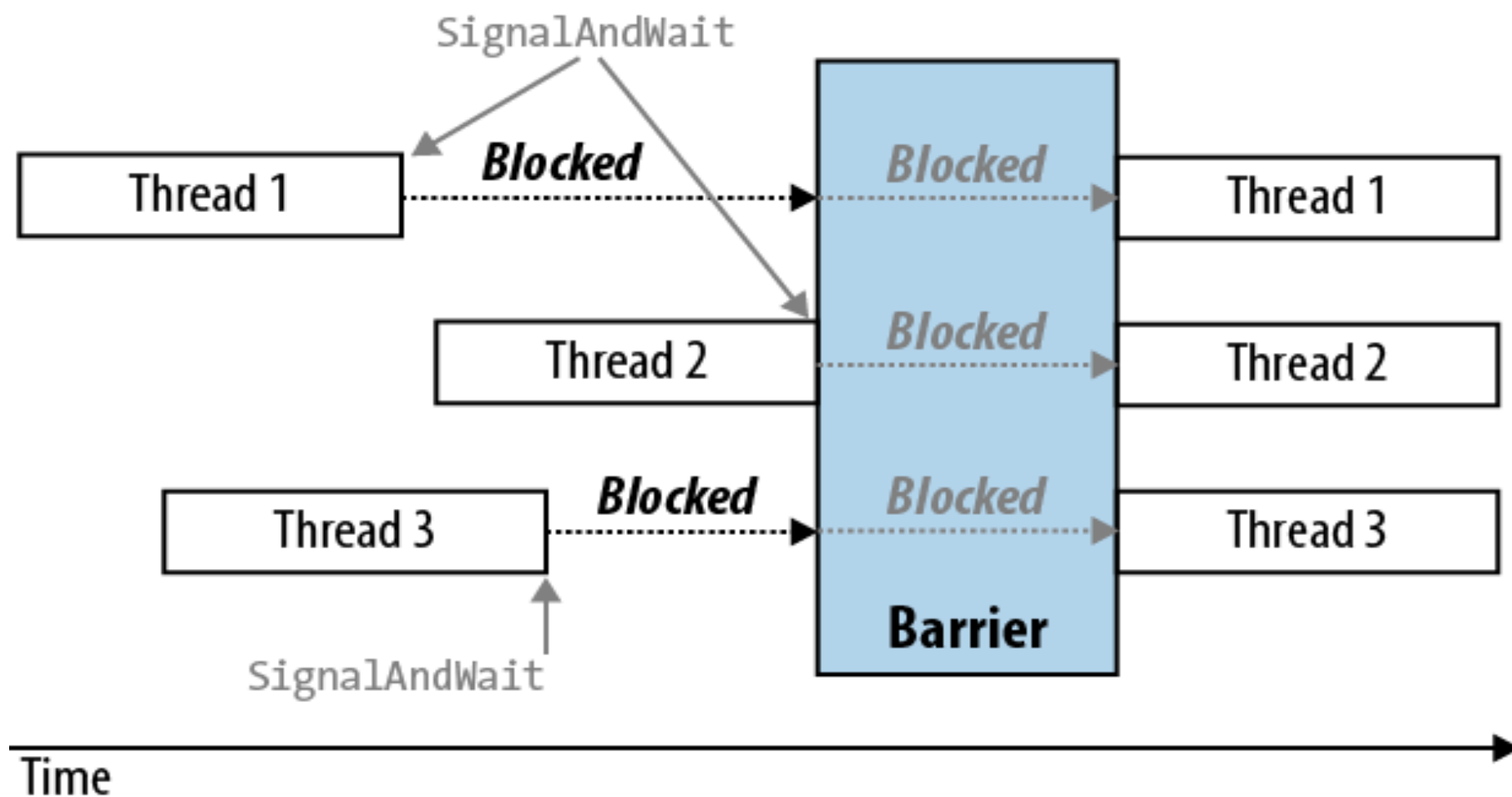
1 1 1

2 2 2

3 3 3

4 4 4

Класс Barrier



Локальное хранилище потока

Иногда данные должны храниться изолированно, гарантируя, что каждый поток имеет отдельную их копию

Расширение локальных данных – локальное хранилище потока

Основное применение локального хранилища потока касается сохранения данных, с помощью которых осуществляется поддержка пути выполнения (обмен сообщениями, транзакции, маркеры безопасности)

```
[ThreadStatic]
static int _x; //не работает с полями экземпляра

void Main()
{
    new Thread (() => { Thread.Sleep(1000); _x++; _x.Dump(); }).Start();
    new Thread (() => { Thread.Sleep(2000); _x++; _x.Dump(); }).Start();
    new Thread (() => { Thread.Sleep(3000); _x++; _x.Dump(); }).Start();
}
```

Локальное хранилище потока

```
static ThreadLocal<int> _x = new ThreadLocal<int> (() => 3); //4.0 + поля!

void Main()
{
    new Thread (() => { Thread.Sleep(1000); _x.Value++; _x.Dump(); }).Start();
    new Thread (() => { Thread.Sleep(2000); _x.Value++; _x.Dump(); }).Start();
    new Thread (() => { Thread.Sleep(3000); _x.Value++; _x.Dump(); }).Start();
}
```

Локальное хранилище потока. GetData и SetData

Методы GetData и SetData сохраняют данные в «ячейках», специфичных для потока. Используют объект LocalDataStoreSlot для идентификации ячейки. Одна и та же ячейка может использоваться во всех потоках, но все они будут получать отдельные значения

Таймеры

Самый простой способ выполнять метод периодически – использовать таймер, класс `Timer` из пространства имен `System.Threading`. Этот таймер использует пул потоков, допуская создание множества таймеров без накладных расходов в виде такого же количества потоков

```
public sealed class Timer : MarshalByRefObject, IDisposableable
{
    public Timer(TimerCallback tick, object state, 1st, subsequent);
    public bool Change(1st, subsequent); // Для изменения периода
    public void Dispose();                // Для удаления
}
// 1st = время до первого срабатывания в миллисекундах или как TimeSpan
// subsequent = следующие интервалы в миллисекундах или как TimeSpan
// (используйте Timeout.Infinite для одноразового срабатывания)
```

Таймеры

```
using System.Threading;
...
static void Main()
{
    // First interval = 5000ms; subsequent intervals = 1000ms
    Timer tmr = new Timer (Tick, "tick...", 5000, 1000);
    Console.WriteLine ("Press Enter to stop");
    Console.ReadLine();
    tmr.Dispose();           // This both stops the timer and cleans up.
}

static void Tick (object data)
{
    // This runs on a pooled thread
    Console.WriteLine (data);           // Writes "tick..."
}
```

Таймеры

.NET Framework предоставляет также другой класс таймера с тем же самым именем в пространстве имен **System.Timers**. Это простая обертка **System.Threading.Timer**, с тем же самым основным механизмом, обеспечивающая дополнительные удобства при использовании пула потоков:

- Реализация в виде компонента, позволяющая размещать его в дизайнере Visual Studio
- Свойство Interval вместо метода Change
- Событие Elapsed вместо делегата обратного вызова
- Свойство Enabled для запуска и остановки таймера (значение по умолчанию – false)
- Методы Start и Stop на случай, если вам не нравится Enabled
- Флаг AutoReset для указания необходимости периодических срабатываний (значение по умолчанию – true)

System.Windows.Forms, System.Web.UI, System.Windows.Threading

Таймеры

```
using System.Timers;
...
static void Main(string[] args)
{
    var tmr = new Timer {Interval = 1000}; // Конструктор без параметров
    tmr.Elapsed += tmr_Elapsed; // Событие вместо делегата
    tmr.Start(); // Запустить таймер
    Console.ReadLine();
    tmr.Stop(); // Остановить таймер
    Console.ReadLine();
    tmr.Start(); // Продолжить
    Console.ReadLine();
    tmr.Dispose(); // Остановить навсегда
}

private static void tmr_Elapsed(object sender, EventArgs e)
{
    Console.WriteLine("Tick");
}
```

- **Async Programing Model (APM)** основан на асинхронных делегатах `BeginInvoke/EndInvoke`, которые используют пару методов с именами *BeginOperationName* и *EndOperationName*, которые соответственно начинают и завершают асинхронную операцию *OperationName*. После вызова метода *BeginOperationName* приложение может продолжить выполнение инструкций в вызывающем потоке, пока асинхронная операция выполняется в другом. Для каждого вызова метода *BeginOperationName* в приложении также должен присутствовать вызов метода *EndOperationName*, чтобы получить результаты операции.
- Данный подход можно встретить во множестве технологий и классов, но он чреват усложнением и избыточностью кода.

Асинхронные делегаты

Асинхронные делегаты предлагают удобный механизм возвращения данных из потока, когда он завершит выполнение, позволяя передавать в обоих направлениях любое количество параметров с контролем их типа

Необработанные исключения из асинхронных делегатов удобно перебрасываются в исходный поток и, таким образом, не требуют отдельной обработки

Асинхронные делегаты также предоставляют альтернативный путь к пулу потоков

```
static void ComparePages()  
{  
    WebClient wc = new WebClient();  
    string s1 = wc.DownloadString("http://www.rsdn.ru");  
    string s2 = wc.DownloadString("http://rsdn.ru");  
    Console.WriteLine(s1 == s2 ? "Одинаковые" : "Различные");  
}
```

Синхронная модель

DownloadString блокирует вызывающий метод на время загрузки страницы (альтернатива **DownloadStringAsync**)

- Говорим DownloadString начать выполнение
- Исполняем другие задачи, пока она работает, например загружаем вторую страницу
- Запрашиваем DownloadString результаты

Асинхронные делегаты

```
static void ComparePages()
{
    Func<string, string> download1 = new WebClient().DownloadString;
    Func<string, string> download2 = new WebClient().DownloadString;

    IAsyncResult cookie1 = download1.BeginInvoke("http://www.rsdn.ru", null, null);
    // Метод обратного вызова и объект с данными (null)!
    IAsyncResult cookie2 = download2.BeginInvoke("http://rsdn.ru", null, null);

    // Выполняем какие-то вычисления:
    Thread.Sleep(5000);

    string s1 = download1.EndInvoke(cookie1); // ожидает, когда делегат завершит
свою работу
    string s2 = download2.EndInvoke(cookie2);

    Console.WriteLine(s1 == s2 ? "Одинаковые" : "Различные");
}
```

Вызывающий поток стыкуется с рабочим, чтобы получить результаты и позволить повторную генерацию исключений (более полезно, чем ThreadPool.QueueWorkerItem или BackgroundWorker)

Асинхронные делегаты

Способы для ожидания результатов от асинхронного делегата

- С помощью `IAsyncResult` можно извлекать информацию о делегате и проверять, завершил ли он свою работу с применением свойства `IsComplete`
- Метод `EndInvoke` сам ожидает, когда делегат завершит свою работу
- Применение дескриптора ожидания (wait handle), который ассоциируется с `IAsyncResult` (`WaitHandle AsyncWaitHandle { get; }` метод `WaitOne()`)
- Применение асинхронного обратного вызова (asynchronous callback)

```
public interface IAsyncResult
{
    object AsyncState { get; }
    WaitHandle AsyncWaitHandle { get; }
    bool CompletedSynchronously { get; }
    bool IsCompleted { get; }
}
```

```
public delegate void AsyncCallback(IAsyncResult ar)
```

Асинхронные делегаты. Асинхронный обратный вызов

Вместо опроса делегата о том, завершился ли асинхронно вызванный метод, было бы более эффективно заставить вторичный поток информировать вызывающий поток о завершении выполнения задания. Чтобы включить такое поведение, необходимо передать экземпляр делегата `System.AsyncCallback` в качестве параметра методу `BeginInvoke`. Если передается объект `AsyncCallback`, делегат автоматически вызовет указанный метод по завершении асинхронного вызова.

Метод обратного вызова будет вызван во вторичном потоке, а не в первичном. Это имеет важное последствие для потоков с графическим интерфейсом пользователя (WPF или Windows Forms), поскольку элементы управления привязаны к потоку, который их создал, и могут управляться только им

Целевые методы `AsyncCallback` должны иметь следующую сигнатуру

```
Void SomeAsyncCallbackMethod(IAsyncResult res)
```

```
public delegate void AsyncCallback(IAsyncResult ar)
```

Асинхронные делегаты. Асинхронный обратный вызов

```
bo.BeginInvoke(1, 5000, TakesAWhileCompleted, bo);
```

```
static void TakesAWhileCompleted(IAsyncResult ar)
{
    if (ar == null)
        throw new ArgumentNullException("ar");

    var bo = ar.AsyncState as BinaryOp;
    if (bo == null) return;
    int result = bo.EndInvoke(ar);
    Console.WriteLine("Результат: " + result);
}
```

пример

- **Event-based Asynchronous Pattern** (примером может служить реализация `BackgroundWorker`);
- Класс, поддерживающий асинхронную модель, основанную на событиях, будет содержать один или несколько методов *MethodNameAsync*. Он может отражать синхронные версии, которые выполняют то же действие с текущим потоком. Также в этом классе может содержаться событие *MethodNameCompleted* и метод *MethodNameAsyncCancel* (или просто *CancelAsync*) для отмены операции. Данный подход распространен при работе с сервисами. Обработка исключений и результаты асинхронной операции доступны только в обработчике события посредством соответствующих свойств параметра: *Error* и *Result*.

BackgroundWorker

BackgroundWorker - класс

BackgroundWorker - конструктор

BackgroundWorker - методы

CancelAsync - метод

Dispose - метод

MemberwiseClone - метод

OnDoWork - метод

OnProgressChanged - метод

OnRunWorkerCompleted - метод

ReportProgress - метод

RunWorkerAsync - метод

BackgroundWorker - свойства

CancellationPending - свойство

IsBusy - свойство

WorkerReportsProgress - свойство

WorkerSupportsCancellation - свойство

BackgroundWorker - события

DoWork - событие

ProgressChanged - событие

RunWorkerCompleted - событие

Чтобы запустить занимающую много времени операцию в фоновом режиме, следует создать экземпляр BackgroundWorker и отслеживать события, сообщающие о ходе выполнения операции и сигнализирующие о ее завершении

Чтобы настроить выполнение операции в фоновом режиме, необходимо добавить обработчик события для события DoWork (вызвать операцию, которая занимает много времени, в этом обработчике событий)

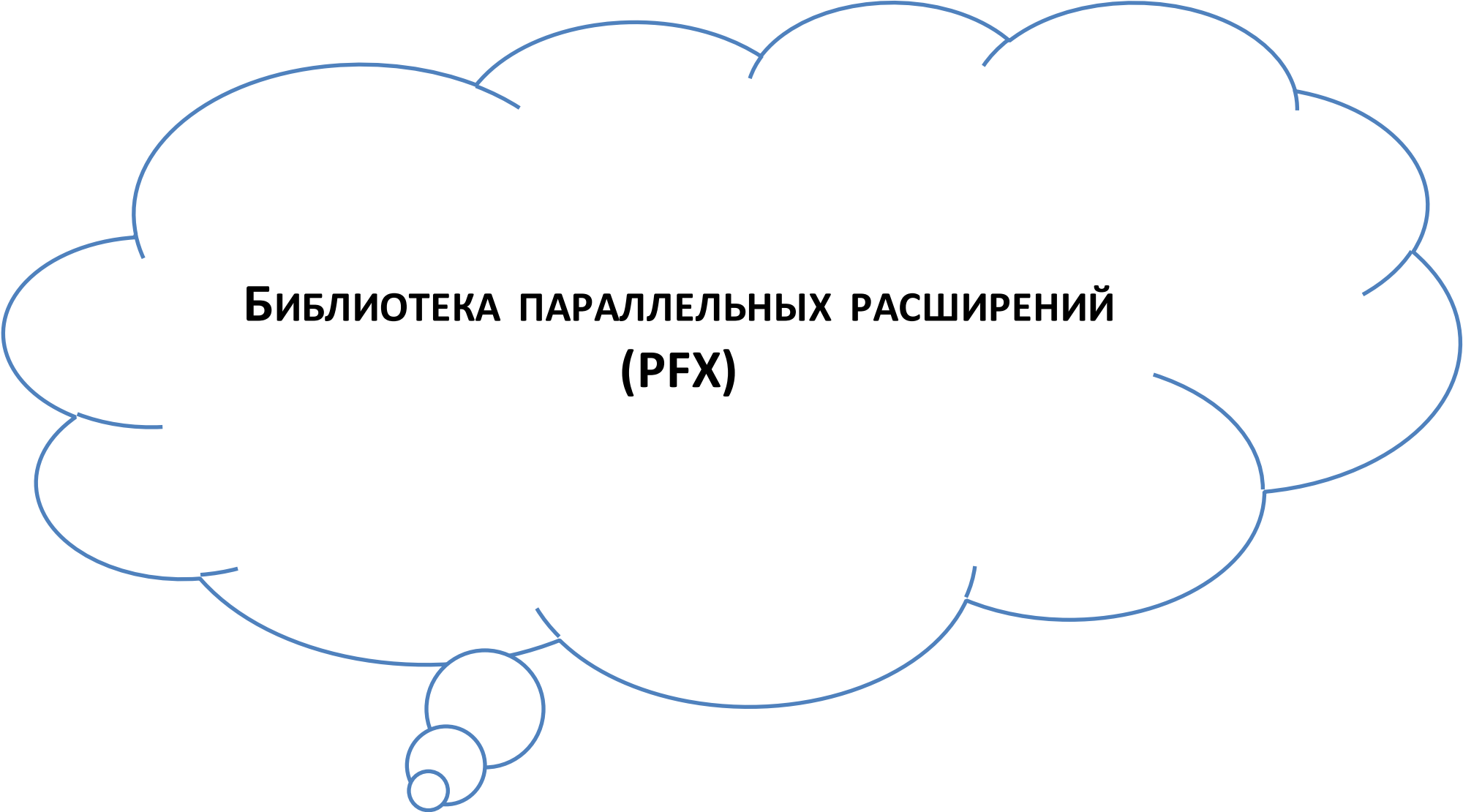
Чтобы начать операцию, вызвать RunWorkerAsync

Чтобы получать уведомления о ходе выполнения, необходимо обработать событие ProgressChanged

Если необходимо получить уведомление после завершения операции, обработать событие RunWorkerCompleted

В .NET Framework 4.0 была введена усовершенствованная модель под названием **TAP (*Task-based Asynchronous Model*)**, которая основана на задачах. Данная реализация асинхронной модели базируется на типах *Task* и *Task<TResult>* *System.Threading.Tasks*, которые используются для предоставления произвольных асинхронных операций. TAP — это рекомендуемый асинхронный шаблон для разработки новых компонентов. Очень важно понимать разницу между потоком (*Thread*) и задачей (*Task*), которые сильно отличаются. *Thread* (поток) представляет собой инкапсуляцию потока выполнения, в то время как *Task* является работой (или просто асинхронной операцией), которая может быть выполнена параллельно. Для выполнения задачи используется свободный поток из пула потоков. По завершении работы поток будет возвращен обратно в пул, а пользователь класса получит результат задачи. Если нужно запустить длительную операцию, не блокируя надолго ни один из потоков пула, то можете это сделать при помощи параметра *TaskCreationOptions.LongRunning*.

В последних версиях фреймворка появились новые возможности на основе все тех же задач, которые упрощают написание асинхронного кода и делают его более читабельным и понятным. Для этого введены новые ключевые слова *async* и *await*, которыми помечаются асинхронные методы и их вызовы. Асинхронный код становится очень похожим на синхронный: необходимо просто вызывать нужную операцию и весь код, который следует за ее вызовом, автоматически будет завернут в некий «колбек», который вызовется после завершения асинхронной операции. Также данный подход позволяет обрабатывать исключения в синхронной манере; явно дожидаться завершения операции; определять действия, которые должны быть выполнены, и соответствующие условия. Например, можно добавить код, который будет выполнен только в том случае, если в асинхронной операции было сгенерировано исключение.



БИБЛИОТЕКА ПАРАЛЛЕЛЬНЫХ РАСШИРЕНИЙ (PFX)

Подходы разработки асинхронных программ

- Threads;
- Async Programing Model (APM представлена с помощью вызова асинхронных методов делегата BeginInvoke/EndInvoke);
- Event-based Asynchronous Pattern (примером может служить реализация BackgroundWorker);
- ThreadPool.QueueUserWorkItem.

Концепции библиотеки Parallel Framework (PFX)

Библиотека параллельных расширений (Parallel Extensions) для создания многопоточных приложений позволяет

- автоматически масштабировать выполняемые задачи, подстраиваясь под фактическое число процессорных ядер
- предлагается упрощенный программный интерфейс (новый набор классов) для конструирования многопоточных приложений

Существует две стратегии разделения работы между потоками: параллелизм данных (data parallelism) и параллелизм задач (task parallelism)

Параллельное программирование (parallel programming) – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).

Концепции библиотеки Parallel Framework (PFX)



Концепции библиотеки Parallel Framework (PFX)

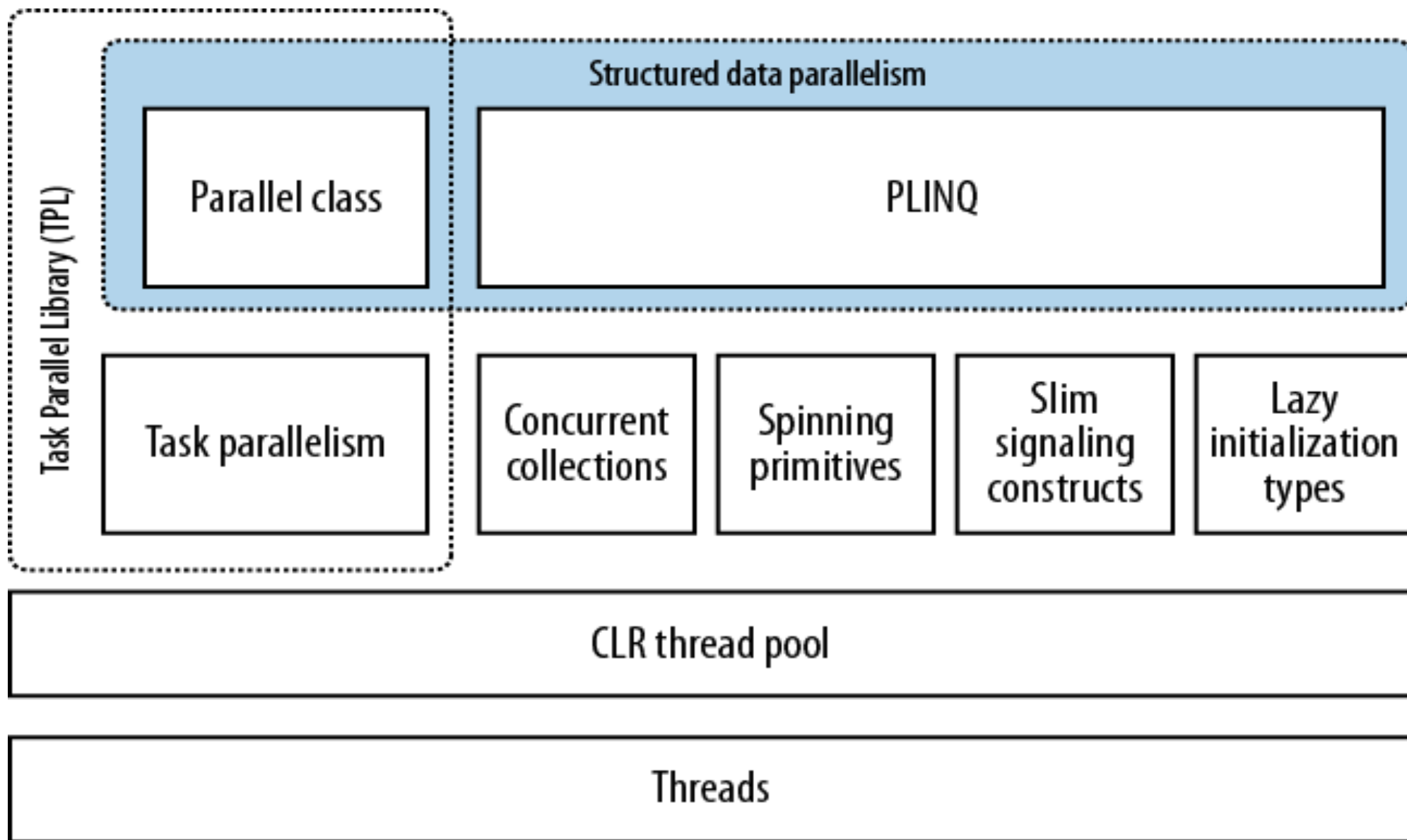
Parallel Extensions обеспечивает три уровня организации параллелизма

Параллелизм на уровне задач. Библиотека обеспечивает высокоуровневую работу с пулом потоков, позволяя явно структурировать параллельно исполняющийся код с помощью легковесных задач. Планировщик библиотеки выполняет диспетчеризацию задач, а также предоставляет единообразный механизм отмены задач и обработки исключительных ситуаций

Параллелизм при императивной обработке данных. Библиотека содержит параллельные реализации основных итеративных операторов, таких как циклы `for` и `foreach`. При этом выполнение автоматически распределяется на все доступные ядра/процессоры вычислительной системы

Параллелизм при декларативной обработке данных - реализуется при помощи параллельного языка интегрированных запросов (PLINQ). PLINQ выполняет запросы параллельно, обеспечивая масштабируемость и загрузку доступных ядер и процессоров

Концепции библиотеки Parallel Framework (PFX)



Концепции библиотеки Parallel Framework (PFX)

Библиотека PLINQ автоматизирует все этапы распараллеливания вычислений, включая разделение работы на задачи, выполнение этих задач различными потоками и объединение результатов в одну выходную последовательность. Ее использование называется декларативным (declarative). В противоположность этому, другие подходы являются императивными: в этом случае вам нужно явно написать код по разделению задачи и объединению результатов. В случае класса `Parallel` необходимо объединить результаты самостоятельно; в случае конструкций параллелизма задач - еще и самостоятельно разделить задачу

	Partitions work	Collates results
PLINQ	Yes	Yes
The <code>Parallel</code> class	Yes	No
PFX's task parallelism	No	No

Параллельные коллекции (concurrent collections) и спин-примитивы (spinning primitives) помогают в решении низкоуровневых задач параллельного программирования

Концепции библиотеки Parallel Framework (PFX)

Некоторые конструкции параллельного программирования также иногда применяются в классических задачах многопоточности:

- PLINQ и класс Parallel полезны, когда нужно выполнить операции параллельно и дождаться окончания их выполнения (структурный параллелизм, structured parallelism). Это включает задачи, не требующие нагрузки центрального процессора, такие как вызов Web-сервиса
- Конструкции параллелизма задач полезны, когда нужно выполнить некоторую операцию в потоке из пула потоков, а также управлять последовательностью выполняемых действий с помощью продолжений (continuations) и родительских/дочерних задач
- Параллельные коллекции иногда полезны, когда нужна потокобезопасная очередь, стек или словарь

Параллелизм на уровне задач

Потоки - это низкоуровневый инструмент для организации параллельных вычислений, обладает следующими ограничениями

- не существует простого способа получить возвращаемое значение обратно из потока, для которого выполняется Join (только разделяемое поле)
- после завершения поток нельзя запустить как-то по другому (можно только присоединится)

как следствие, препятствие мелкомодульному параллелизму

прямое использование потоков - производительность

```
// Task is in System.Threading.Tasks
Task.Run (() => Console.WriteLine ("Hello from the thread pool"));

ThreadPool.QueueUserWorkItem (notUsed => Console.WriteLine ("Hello"));
```

Параллелизм на уровне задач

Параллелизм на уровне задач базовый уровень библиотеки Parallel Extensions. Task (задача) - абстракция, представляющей асинхронную операцию, которая может быть подкреплена или не подкреплена потоком

Исполнением задач управляет специальный планировщик задач, учитывающий фактическое число процессорных ядер (несколько задач могут разделять один и тот же поток) (пул потоков) Для представления задач используются классы Task и Task<T> пространстве имен System.Threading.Tasks

Класс	Назначение
Task	Для управления единицей работы
Task<TResult>	Для управления единицей работы, которая возвращает значение
TaskFactory	Для создания задач
TaskFactory<TResult>	Для создания задач и продолжений с тем же типом возвращаемого значения
TaskScheduler	Для управления планировщиком задач
TaskCompletionSource	Для ручного управления жизненным циклом задачи

Параллелизм на уровне задач

Имя элемента	Описание
AsyncState	Объект, заданный при создании задачи как аргумент Action<object>
ContinueWith(), ContinueWith<T>()	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
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>
Id	Целочисленный идентификатор задачи
IsCanceled	Булево свойство, указывающее, была ли задача отменена
IsCompleted	Свойство равно true, если выполнение задачи успешно завершилось
IsFaulted	Свойство равно true, если задача сгенерировала исключение
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

Параллелизм на уровне задач. Создание и запуск задачи

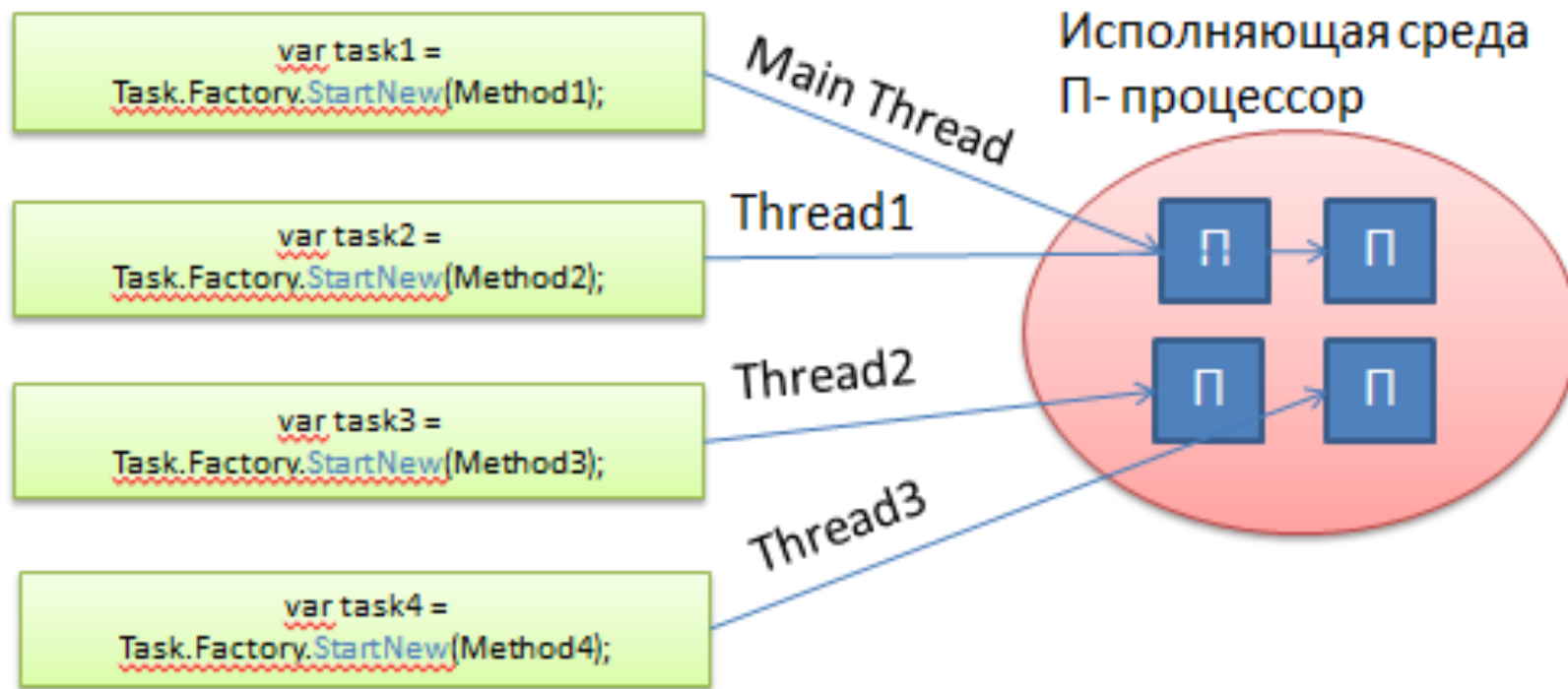
Для создания задачи используется один из перегруженных конструкторов класса Task

```
public Task(Action действие)
```

действие обозначает точку входа в код, представляющий задачу, тогда как Action — делегат

```
Action work = () =>
{
    Thread.Sleep(2000);
    Console.WriteLine("Done");
};
Action<object> work2 = obj =>
{
    Thread.Sleep(1000);
    Console.WriteLine(obj.ToString());
};
var t1 = new Task(work);
var t2 = new Task(work2, 25);
```

Параллелизм на уровне задач. Создание и запуск задачи



Интересная особенность класса Task заключается в том, что он спроектирован так, чтобы распределять нагрузку по выполняемым задачам равномерно по процессорам в системе

Параллелизм на уровне задач. Создание и запуск задачи

Созданная задача ставится в очередь планировщика для запуска при помощи методов `Start` или `RunSynchronously` (запускает задачу в текущем потоке), могут принимать аргумент типа `TaskScheduler` (пользовательский планировщик задач)

```
t1.Start();                // асинхронный запуск
t2.RunSynchronously();     // синхронный запуск
Console.WriteLine("Task started"); // напечатано через 1 сек
```

После того, как задача завершена, она не может быть перезапущена. Следовательно, иного способа повторного запуска задачи на исполнение, кроме создания ее снова, не существует!

Можно создать и запустить `Task`, вызвав метод `Task.Factory.StartNew` (создает и запускает задачу за один шаг) и передав в него делегат `Action`

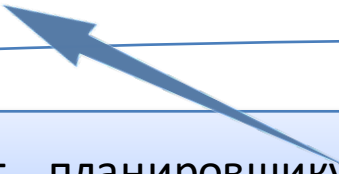
```
Task.Factory.StartNew (() => Console.WriteLine ("Hello from a task!"));
```

Параллелизм на уровне задач. TaskCreationOptions

Процесс выполнения задачи можно настроить путем использования перечисления TaskCreationOptions во время вызова метода StartNew (или создания экземпляра класса Task). TaskCreationOptions – это флаговое перечисление со следующими (объединяемыми) значениями:

- LongRunning
- PreferFairness
- AttachedToParent

```
var t1 = new Task(work, TaskCreationOptions.LongRunning);
```



Говорит планировщику выделить для задачи отдельный поток

PreferFairness говорит планировщику попытаться распределить задачи в том же порядке, в котором они были запущены. Обычно это не так, поскольку планировщик оптимизирует работу задач путем использования локальных очередей

Параллелизм на уровне задач. TaskCreationOptions

```
Task parent = Task.Factory.StartNew (() =>
{
    Console.WriteLine ("I am a parent");

    Task.Factory.StartNew (() =>           // Независимая задача
    {
        Console.WriteLine ("I am detached");
    });

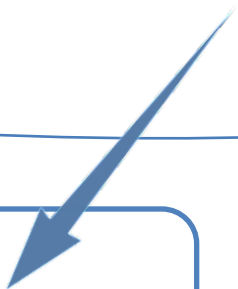
    Task.Factory.StartNew (() =>           // Дочерняя задача
    {
        Console.WriteLine ("I am a child");
    }, TaskCreationOptions.AttachedToParent);
});
```

Предназначено для создания дочерних задач

Параллелизм на уровне задач. Возврат значений

```
Task<int> task = Task.Run (() => { Console.WriteLine ("Foo"); return 3; });  
  
int result = task.Result;           // Blocks if not already finished
```

```
Task<int> primeNumberTask = Task.Run (() =>  
    Enumerable.Range (2, 3000000)  
        .Count (n => Enumerable.Range(2, (int)Math.Sqrt(n)-1)  
            .All (i => n % i > 0)));  
  
Console.WriteLine ("Task running...");  
Console.WriteLine ("The answer is " + primeNumberTask.Result);
```



Task running...
The answer is 216816

Параллелизм на уровне задач. Ожидание задач

Можно явно ожидать завершения задачи двумя способами:

- Путем вызова метода `Wait` (с опциональным указанием тайм-аута).
- Путем обращения к свойству `Result` (`Task<TResult>`)

Можно одновременно ожидать завершения нескольких задач с помощью статических методов `Task.WaitAll` (ожидать завершения всех указанных задач) и `Task.WaitAny` (ожидать завершения какой-либо задачи)

В `Wait`-методы также можно передать `CancellationToken` (маркер отмены)

```
t1.Wait(1000);  
Task.WaitAll(t1, t2);
```

Параллелизм на уровне задач. Класс TaskCompletionSource

Класс TaskCompletionSource позволяет создавать задачу из любой операции, которая начинается и заканчивается некоторое время спустя. Он работает путем предоставления «подчиненной» задачи, которая управляется вручную, указывая, когда задача завершилась или отказала

Экземпляр класса TaskCompletionSource предоставляет свойство Task, которое возвращает задачу, завершение выполнения которой можно ожидать или присоединять к ней продолжения. Задача полностью контролируется с помощью объекта TaskCompletionSource посредством его методов

```
public class TaskCompletionSource<TResult>
{
    public void SetResult(TResult result);
    public void SetException(Exception exception);
    public void SetCanceled();

    public bool TrySetResult(TResult result);
    public bool TrySetException(Exception exception);
    public bool TrySetCanceled();
    ...
}
```

Параллелизм на уровне задач. Класс TaskCompletionSource

```
var tcs = new TaskCompletionSource<int>();  
new Thread (() => { Thread.Sleep (5000); tcs.SetResult (42); }).Start();  
Task<int> task = tcs.Task;           // Our "slave" task.  
Console.WriteLine (task.Result);    // 42
```

```
void Main()  
{  
    Task<int> task = Run (() => { Thread.Sleep (5000); return 42; });  
    Console.WriteLine (task.Result);    // 42  
}
```

```
Task<TResult> Run<TResult> (Func<TResult> function)  
{  
    var tcs = new TaskCompletionSource<TResult>();  
    new Thread (() =>  
        {  
            try { tcs.SetResult (function()); }  
            catch (Exception ex) { tcs.SetException (ex); }  
        }).Start();  
    return tcs.Task;  
}
```

Параллелизм на уровне задач. Обработка ошибок в задачах

При ожидании завершения задачи (либо путем вызова метода `Wait`, либо путем доступа к свойству `Result`) любое необработанное исключение будет передано вызывающему коду, обернутое в объект `AggregationException`

```
int x = 0;
Task<int> calc = Task.Factory.StartNew (() => 7 / x);
try
{
    Console.WriteLine (calc.Result);
}
catch (AggregateException aex)
{
    Console.Write (aex.InnerException.Message); // Пытались разделить на 0
}
```

Устраняет необходимость в написании кода обработки непредвиденных исключений внутри задачи

Параллелизм на уровне задач. Обработка ошибок в задачах

```
// Start a Task that throws a NullReferenceException:
//Task task = Task.Run (() => { throw null; });
Task task = Task.Factory.StartNew (() => { throw null; });
try
{
    task.Wait();
}
catch (AggregateException aex)
{
    if (aex.InnerException is NullReferenceException)
        Console.WriteLine ("Null!");
    else
        throw;
}
```

При выполнении метода `Wait` могут быть сгенерированы два исключения: `ObjectDisposedException` (генерируется в том случае, если задача освобождена посредством вызова метода `Dispose`) и `AggregateException` (генерируется в том случае, если задача сама генерирует исключение или же отменяется)

Свойства `IsFaulted` и `IsCanceled`

Параллелизм на уровне задач. Обработка ошибок в задачах

Ожидание завершения родительских задач приводит к ожиданию всех дочерних задач и к обработке исключений, возникших в этих задачах

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
var parent = Task.Factory.StartNew (() =>
{
    Task.Factory.StartNew (() =>    // Child
    {
        Task.Factory.StartNew (() => { throw null; }, atp);    // Grandchild
    }, atp);
});

// The following call throws a NullReferenceException (wrapped
// in nested AggregateExceptions):
parent.Wait();
```

Класс `AggregateException` предоставляет пару методов, упрощающих обработку исключений: методы `Flatten` и `Handle`

Параллелизм на уровне задач. Продолжения

Иногда бывает полезно запустить на выполнение задачу сразу же после завершения выполнения другой задачи (или после неудачного завершения этой задачи). Метод `ContinueWith` класса `Task` предоставляет именно такую возможность

```
Task task1 = Task.Factory.StartNew (() => Console.Write ("antecedant.."));  
Task task2 = task1.ContinueWith (ant => Console.Write ("..continuation"));
```

Как только задача `task1` (родительская задача) будет выполнена, неудачно завершится или будет отменена, задача `task2` (продолжение) запустится автоматически

Может быть особенно важным, если задача `task1` возвращает данные!

Параллелизм на уровне задач. Продолжения и Task<TResult>

Продолжения могут иметь тип Task<TResult> и возвращать некоторые данные

```
Task.Factory.StartNew<int> (() => 8)
    .ContinueWith (ant => ant.Result * 2)
    .ContinueWith (ant => Math.Sqrt (ant.Result))
    .ContinueWith (ant => Console.WriteLine (ant.Result));    // 4
```

В реальных приложениях, лямбда-выражения будут вызывать функции с интенсивными вычислениями

Параллелизм на уровне задач. Отмена выполнения задач

При запуске задачи можно передать маркер отмены, что позволяет отменить выполнение задачи

```
var cancelSource = new CancellationTokenSource();
CancellationToken token = cancelSource.Token;
cancelSource.Cancel();
Task task = Task.Factory.StartNew (() =>
{
    // Выполняем некоторые операции...
    token.ThrowIfCancellationRequested(); // Проверяем запрос отмены
    // Выполняем некоторые операции...
}, token);
...
```

Сначала получается признак отмены (CancellationToken) из источника признаков отмены (CancellationTokenSource). Затем этот признак передается задаче, после чего она должна контролировать его на предмет получения запроса на отмену. (Этот запрос может поступить только из источника признаков отмены.) Если получен запрос на отмену, задача должна завершиться

Параллелизм на уровне задач. Отмена выполнения задач

```
var cts = new CancellationTokenSource();
CancellationToken token = cts.Token;
cts.CancelAfter(500);
Task task = Task.Factory.StartNew(() =>
{
    Thread.Sleep(1000);
    if(token.IsCancellationRequested)
        token.ThrowIfCancellationRequested(); // Check for cancellation request
}, token);
```

```
try
{
    task.Wait();
}
catch (AggregateException ex)
{
    if (ex.InnerException is OperationCanceledException)
        Console.WriteLine("Task canceled!");
}
```

PLINQ (Parallel Language-Integrated Query) - параллельная реализация LINQ, в которой запросы выполняются параллельно, используя все доступные ядра и процессоры. PLINQ полностью поддерживает все операторы запросов, имеющиеся в LINQ to Objects, и имеет минимальное влияние на существующую модель LINQ-операторов

PLINQ

PLINQ автоматически распараллеливает локальные LINQ-запросы, он берет на себя разбиение задачи и объединения результатов

//3апрос вычисляет простые числа от 3 до 100 000, используя все ядра процессора

```
IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);  
  
var parallelQuery =  
    from n in numbers.AsParallel()  
    where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)  
    select n;  
  
int[] primes = parallelQuery.ToArray();
```

System.Linq.ParallelEnumerable



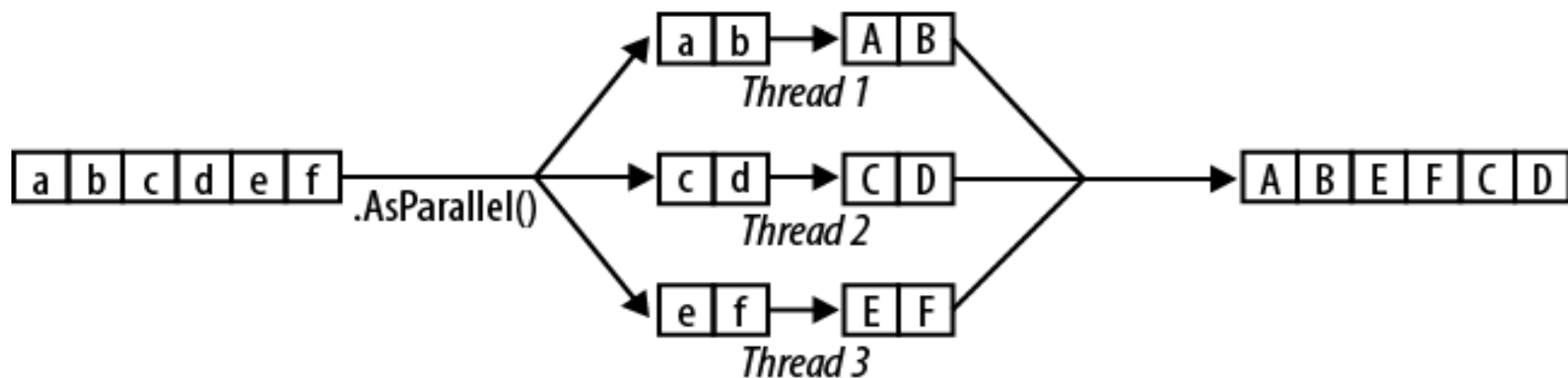
Оборачивает входные данные в последовательность, построенную на основе ParallelQuery<TSource>, что приводит к тому, что операторы запросов LINQ будут вызывать альтернативный набор методов расширения, определенные в классе ParallelEnumerable

PLINQ

```
namespace System.Linq
{
    ... public static class ParallelEnumerable
    {
        ... public static TSource Aggregate<TSource>(this ParallelQuery<TSource>
        ... public static TAccumulate Aggregate<TSource, TAccumulate>(this Parallel
        ... public static TResult Aggregate<TSource, TAccumulate, TResult>(this F
        ... public static TResult Aggregate<TSource, TAccumulate, TResult>(this F
        ... public static TResult Aggregate<TSource, TAccumulate, TResult>(this F
        ... public static bool All<TSource>(this ParallelQuery<TSource> source, F
        ... public static bool Any<TSource>(this ParallelQuery<TSource> source);
        ... public static bool Any<TSource>(this ParallelQuery<TSource> source, F
        ... public static IEnumerable<TSource> AsEnumerable<TSource>(this Paralle
        ... public static ParallelQuery<TSource> AsOrdered<TSource>(this Parallel
        ... public static ParallelQuery AsOrdered(this ParallelQuery source);
        ... public static ParallelQuery<TSource> AsParallel<TSource>(this IEnumerable
        ... public static ParallelQuery AsParallel(this IEnumerable source);
        ... public static ParallelQuery AsParallel(this IEnumerable source, ...
    }
```

Кроме `AsParallel()`, класс `ParallelEnumerable` содержит еще несколько особых методов:

- `AsSequential()` конвертирует объект `ParallelQuery<T>` в коллекцию `IEnumerable<T>` так, что все запросы выполняются последовательно
- `AsOrdered()` при параллельной обработке заставляет сохранять в `ParallelQuery<T>` порядок элементов
- `AsUnordered()` при параллельной обработке позволяет игнорировать в `ParallelQuery<T>` порядок элементов
- `WithCancellation()` устанавливает для `ParallelQuery<T>` указанное значение токена отмены
- `WithDegreeOfParallelism()` устанавливает для `ParallelQuery<T>` целочисленное значение степени параллелизма (число ядер процессоров)
- `WithExecutionMode()` задает опции выполнения параллельных запросов в виде перечисления `ParallelExecutionMode`

ParallelEnumerable.Select

```
"abcdef" .AsParallel().Select (c => char.ToUpper(c)).ToArray()
```

Вызов метода `AsSequential()` производит операцию, обратную `AsParallel()`, то есть преобразует `ParallelQuery<TSource>` в `IEnumerable<T>`

В операторах запросов, принимающих две входящие последовательности (`Join`, `GroupJoin`, `Concat`, `Union`, `Intersect`, `Except` и `Zip`), нужно применять `AsParallel()` к обеим входящим последовательностям (иначе исключение)

Повторный вызов `AsParallel` будет неэффективен, так как приведет к слиянию результатов и повторному распараллеливанию запроса.

```
mySequence.AsParallel()  
    .Where (n => n > 100)  
    .AsParallel()  
    .Select (n => n * n)
```

PLINQ

Не все операторы запросов могут эффективно распараллеливаться. Те операторы, для которых это невозможно, реализуются в PLINQ последовательно. PLINQ может также выполнить оператор последовательно, если он подозревает, что накладные расходы на параллелизм на самом деле замедлят выполнение запроса

PLINQ применим только к локальным коллекциям: он не работает с LINQ to SQL или с Entity Framework

Почему AsParallel не используется по умолчанию?

- Использование PLINQ приносит пользу только при наличии значительного количества вычислительных задач, распределенных по рабочим потокам
- Многие запросы LINQ to Objects выполняются очень быстро, и распараллеливание не только не нужно – накладные расходы на разделение данных, объединения результатов и координацию дополнительных потоков могут, на самом деле, только ухудшить производительность
- Результат работы PLINQ запроса (по умолчанию) может отличаться от результатов выполнения LINQ-запроса порядком выходных элементов
- PLINQ заворачивает исключения в AggregateException (для обработки нескольких возможных сгенерированных исключений)
- PLINQ дает ненадежные результаты, если запросы вызывают потоконебезопасные методы

PLINQ. Пример: параллельная проверка правописания

Реализовать проверку правописания, которая будет быстро обрабатывать очень большие документы и будет использовать все доступные ядра процессора

Класс Parallel

PFX предоставляет базовую форму структурного параллелизма с помощью трех методов класса Parallel:

- **Parallel.Invoke:** выполняет параллельно массив делегатов
- **Parallel.For:** параллельный эквивалент цикла for
- **Parallel.ForEach:** параллельный эквивалент цикла foreach

Все три метода блокируют управление до окончания выполнения всех действий

При возникновении необработанного исключения оставшиеся рабочие потоки прекращают выполнение после завершения обработки текущего элемента, и исключение (или исключения), завернутое в AggregationException, передается вызывающему коду

Класс Parallel. Parallel.Invoke

Метод Parallel.Invoke выполняет массив делегатов типа Action параллельно, и затем ожидает их завершения

```
public static void Invoke(params Action[] actions);
```

Одновременная загрузка двух Web-страниц

```
Parallel.Invoke (  
    () => new WebClient().DownloadFile ("http://www.linqpad.net", "lp.html"),  
    () => new WebClient().DownloadFile ("http://www.jaoo.dk", "jaoo.html"));
```

Метод Parallel.Invoke будет выполняться эффективно, даже если передать ему миллион делегатов. Это связано с тем, что он разбивает большое количество элементов на пачки, которые назначаются набору объектов Task, а не создает по отдельному объекту Task для каждого делегата

Класс Parallel. Parallel.Invoke

При использовании всех методов класса Parallel ответственность за объединение результатов лежит на разработчике (безопасность потоков)

```
var data = new List<string>();  
Parallel.Invoke (  
    () => data.Add (new WebClient().DownloadString ("http://www.foo.com")),  
    () => data.Add (new WebClient().DownloadString ("http://www.far.com")));
```

Код не является потокобезопасным

Решение - потокобезопасная коллекция ConcurrentBag

Класс Parallel. Parallel.Invoke

Существует перегруженная версия метода Parallel.Invoke, которая принимает объект класса ParallelOptions

```
public static void Invoke (ParallelOptions options,  
                           params Action[] actions);
```

С помощью ParallelOptions можно добавить маркер отмены, ограничить максимальное количество рабочих потоков или указать свой планировщик задач (custom task scheduler)

Использование маркеров отмены полезно, когда число одновременно выполняемых задач превосходит (примерное) количество ядер процессора: при отмене все делегаты, выполнение которых еще не было начато, будут отменены, однако все делегаты, выполнение которых уже начато, продолжат до завершения

Класс Parallel. Parallel.For и Parallel.ForEach

Методы Parallel.For и Parallel.ForEach аналогичны C#-операторам цикла for и foreach, за исключением того, что итерирование элементов последовательности происходит параллельно, а не последовательно (упрощенные сигнатуры)

```
public static ParallelLoopResult For(int fromInclusive,  
                                   int toExclusive, Action<int> body)  
  
public static ParallelLoopResult ForEach<TSource>(  
    IEnumerable<TSource> source, Action<TSource> body)
```

Класс Parallel. Parallel.For и Parallel.ForEach

Методы Parallel.For и Parallel.ForEach аналогичны C#-операторам цикла for и foreach, за исключением того, что итерирование элементов последовательности происходит параллельно, а не последовательно. Вот их (упрощенные) сигнатуры

```
for(int i = 0; i < 100; i++)  
    Foo (i);
```

```
Parallel.For (0, 100, i => Foo (i));  
//или  
Parallel.For (0, 100, Foo);
```

Класс Parallel. Parallel.For и Parallel.ForEach

Методы Parallel.For и Parallel.ForEach аналогичны C#-операторам цикла for и foreach, за исключением того, что итерирование элементов последовательности происходит параллельно, а не последовательно.

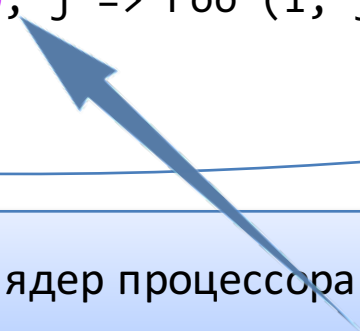
```
foreach(char c in "Hello, world")  
    Foo (c);
```

```
Parallel.ForEach("Hello, world", Foo);
```

Класс Parallel. Внешние циклы vs внутренние циклы

Методы `Parallel.For` и `Parallel.ForEach` обычно лучше работают во внешних, а не во внутренних циклах. Это связано с тем, что в первом случае для распараллеливания вы предоставляете блоки (chunks) большего размера, снижая влияние накладных расходов. Распараллеливать одновременно внешние и внутренние циклы обычно не нужно.

```
Parallel.For(0, 100, i =>
{
    // Для внутреннего цикла лучше использовать последовательное выполнение.
    Parallel.For (0, 50, j => Foo (i, j));
});
```



Понадобится более 100 ядер процессора для получения преимуществ от распараллеливания внутреннего цикла

Класс Parallel. Пример: параллельная проверка правописания

Реализовать проверку правописания, которая будет быстро обрабатывать очень большие документы и будет использовать все доступные ядра процессора

Обработка исключений и отмена выполнения задач

В библиотеке параллельных расширений используются следующие принципы работы с исключительными ситуациями:

- при возникновении исключения в задаче (как созданной явно, так и порожденной неявно, например, методом `Parallel.For()`) это исключение обрабатывается средствами библиотеки (если перехват не был предусмотрен программистом) и перенаправляется в ту задачу, которая ожидает завершения данной
- при параллельном возникновении нескольких исключений все они собираются в единое исключение `System.AggregateException`, которое переправляется дальше по цепочке вызовов задач
- если возникла в точности одна исключительная ситуация, то на ее основе будет создан объект `AggregateException` в целях единообразной обработки всех исключительных ситуаций

Обработка исключений и отмена выполнения задач

Исключительные ситуации типа `AggregateException` могут возникать при работе со следующими конструкциями библиотеки параллельных расширений:

- Класс `Task` - исключения, возникшие в теле задачи, будут повторно возбуждены в месте вызова метода `Wait()` данной задачи. Кроме того, исключение доступно через свойство `Exception` объекта `Task`
- Класс `Task<T>` - исключения, возникшие в теле задачи, будут повторно возбуждены в месте вызова метода `Wait()` или в месте обращения к экземпляру ному свойству `Task<T>.Result`
- Класс `Parallel` - исключения могут возникнуть в параллельно исполняемых итерациях циклов `Parallel.For()` и `Parallel.ForEach()` или в параллельных блоках кода при работе с `Parallel.Invoke()`
- `PLINQ` - из-за отложенного характера исполнения запросов `PLINQ`, исключения обычно возникают на этапе перебора элементов, полученных по запросу

Параллельные коллекции

Framework 4.0 предоставляет набор новых потокобезопасных коллекций в пространстве имен `System.Collections.Concurrent`

Concurrent collection	Nonconcurrent equivalent
<code>ConcurrentStack<T></code>	<code>Stack<T></code>
<code>ConcurrentQueue<T></code>	<code>Queue<T></code>
<code>ConcurrentBag<T></code>	(none)
<code>ConcurrentDictionary<TKey, TValue></code>	<code>Dictionary<TKey, TValue></code>

Параллельные коллекции

- Параллельные коллекции оптимизированы для параллельного программирования. Стандартные коллекции их превосходят во всех случаях, кроме сценариев с высокой конкурентностью
- Потокобезопасные коллекции не гарантируют, что код, который их использует, будет потокобезопасным
- Если в процессе перебора элементов параллельной коллекции другой поток ее модифицирует, исключение сгенерировано не будет. Вместо этого вы получите смесь старого и нового содержимого
- Не существует параллельной версии `List<T>`
- Параллельные классы стека, очереди и набора (bag) внутри реализованы на основе связанных списков. Это делает их менее эффективными в плане потребления памяти по сравнению с непараллельными версиями классов `Stack` и `Queue`, но более предпочтительными для параллельного доступа, поскольку связанные списки являются отличными кандидатами для lock-free или low-lock реализаций (поскольку вставка узла в связанный список требует модификации лишь пары ссылок, в то время как вставка элемента в структуру данных наподобие `List<T>` может потребовать перемещения тысяч существующих элементов)

Спасибо за внимание

БГУ, ММФ, кафедра веб-технологий и компьютерного
моделирования

Автор: к. ф.-м. н., доцент, Кравчук Анжелика Ивановна

e-mail: anzhelika.kravchuk@gmail.com