Информатика

многозадачность

Однозадачность

- На заре компьютерной эры в ОС одновременно могло выполняться только одно задание
- Если приложение зацикливалось, вся система зависала, спасала только перезагрузка

Многозадачность

- Очевидно, необходимы механизмы одновременного выполнения нескольких задач
- Один из подходов – вытесняющая многозадачность
- Выделять задачам кванты времени

I/O bound vs CPU bound

- Вычислительные задачи можно разделить на две категории:
  - **I/O bound** - требующие ввода/вывода
  - **CPU bound** - нагружающие процессор
- I/O bound задачи могут блокироваться, позволяя выполнять работу процессам, нуждающимся в процессорной мощности

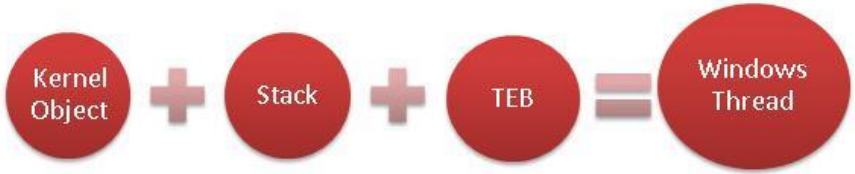
Вычислительные потоки

- Начиная с Windows NT, параллельное выполнение кода возможно с помощью многопоточности
- **Поток (тред, нить) –** блок кода, который может выполняться одновременно с другими потоками
- Потоки – *виртуализация процессора* в Windows

Потоки и процессы

- Поток != Процесс
  - у процесса своё адресное пространство
  - у потоков – разделяемое
  - у процесса есть хотя бы один поток (Main)
  - 1 процесс может содержать несколько потоков

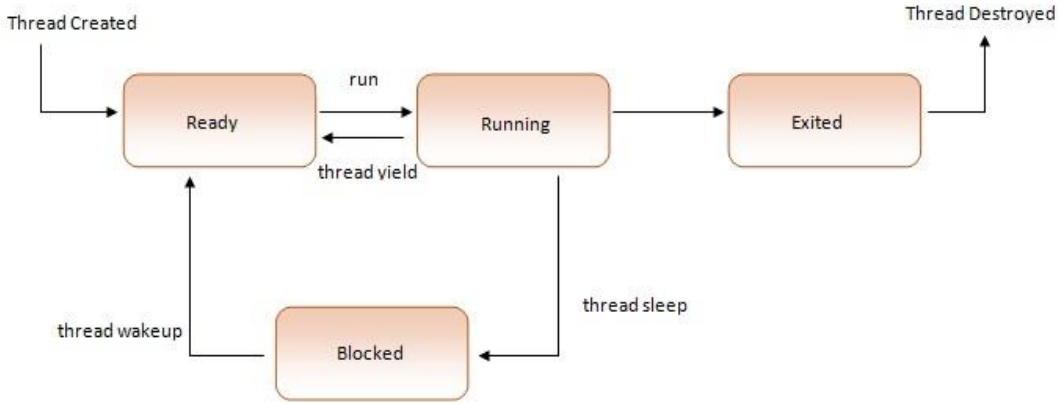
Что содержит поток



- **Thread kernel object**
  - структура данных для данных о потоке
  - регистры процессора, статистика
- **Thread Environment Block** (TEB)
  - данные для обработки исключений
- User-mode **Stack** & Kernel-mode stack (для безопасности)

https://www.codeproject.com/Articles/662735/WebControls/

Жизненный цикл потока



- **Running** – использует процессор
- **Blocked** – ожидает ввод
- **Ready** – готов к запуску
- **Exited** – завершен, но не уничтожен

Потоки в CLR

- Класс Thread пространства имён System.Threading
- CLR потоки *аналогичны Windows-потокам*
  - в первых версиях предполагались логические потоки

Создание потока

- В основном потоке создаётся экземпляр

```
Thread thread = new Thread(WriteY);
```

- В конструкторе указывается метод операции, которая будет выполняться в отдельном потоке
- Вызов метода Start у экземпляра создаёт новый поток

```
thread.Start();
```

Что происходит при создании

- Создаётся Windows поток
- У всех загруженных в процесс DLL-библиотек вызывается DllMain с флагом DLL-THREAD_ATTACH
  - библиотеки C# не имеют DllMain
  - у библиотек может быть отключено получение уведомлений

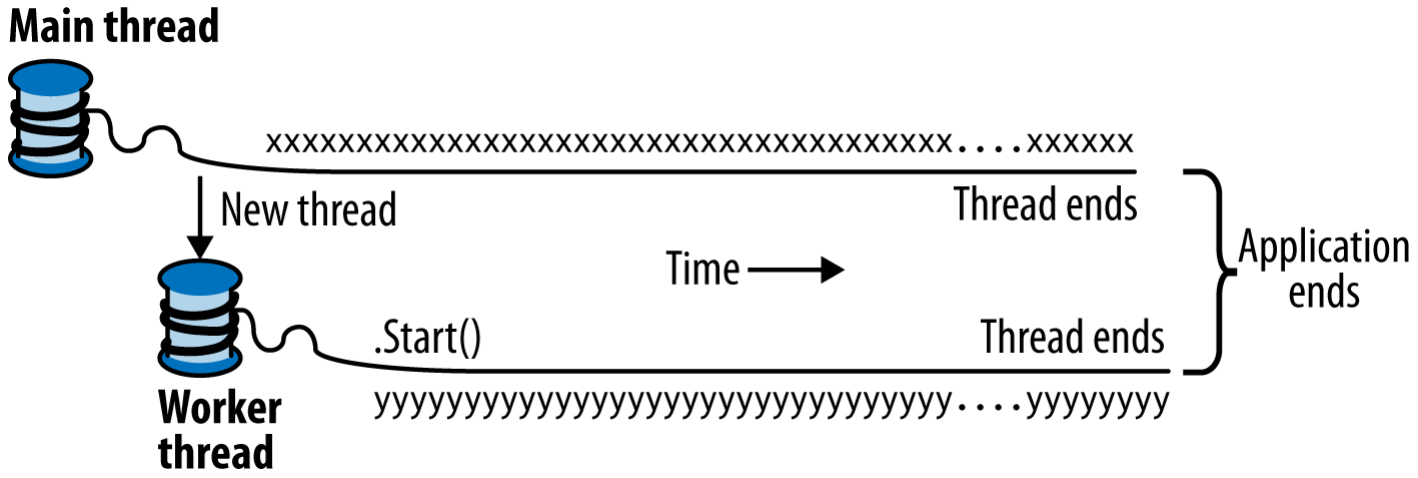
Concurrency

```
static void Main()
{
    Thread thread = new Thread(WriteY);
    thread.Start();
    for (int i = 0; i < 1000; i++) Console.Write("x");
}

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

Одновременное выполнение

- Передача управления происходит *недетерминированно*



Concurrency vs Multithreading

- **Concurrency** – выполнение более одной операции одновременно
- **Multithreading** – использование более одного потока
- ***Многопоточность – способ достижения одновременности выполнения операций***

Параллелизм и переключение

Con – вместе, Current – выполнение

- На многоядерных процессорах потоки могут выполняться одновременно
- Если потоков больше, чем процессоров, происходит переключение
  - выделяются кванты времени 20-30мс

Что происходит при переключении

- Значения регистров процессора сохраняются в контексте ядра потока
- Из набора потоков выделяется тот, которому будет передано управлению.
  Если выбранный поток принадлежит другому процессу, ОС переключает адресное пространство
- Значения из контекста ядра потока загружаются в регистры процессора

Идеальная ситуация

- **Параллельное выполнение**
  - вид одновременного выполнения
  - частный случай использования многопоточности
  - *независимое выполнение потоков на разных ядрах без переключения*

GC, отладка и потоки

- GC при работе
  - приостанавливает все потоки
  - просматривает стеки
  - помечает объекты в куче
  - снова просматривает стеки
    (обновляя перемещенные объекты)
  - возобновляет исполнение всех потоков
- При отладке Windows приостанавливает все потоки в каждой точке останова и на каждом шаге отладки

Сон и передача выполнения

- Можно усыпить текущий поток на указанное в параметре время `Thread.Sleep(1000);`
  - квант времени немедленно отдаётся другому
- Если указать нулевое время `Thread.Sleep(0);` произойдёт переключение на другой поток
- При вызове `Thread.Yield();` произойдёт аналогичная *передача выполнения*,
  но *только потокам того же процессора*

Воссоединение потоков

- При создании потока происходит разделение выполнения
- Можно воссоединить потоки, заставив одного потока ожидать завершения другого

```
var thread = new Thread(Operation);
thread.Start(5);
                ...
thread.Join();
```

Ожидание с таймаутом

- При ожидании текущий поток приостанавливает работу
- Можно указать *таймаут ожидания*
- По окончанию ожидания вернётся `bool`
  - true – поток завершился, false – таймаут

```
bool result = thread.Join(1000);
Console.WriteLine(result?"Дождались":"Устали ждать");
```

Завершение потока

- Можно ***принудительно завершить поток***

```
var thread = new Thread(Operation);
thread.Abort();
```

- При этом в потоке thread возникнет исключение ThreadAbortException

Завершение потока

```
static void Main(string[] args)
{
    var thread = new Thread(Operation);
    thread.Start();
    Console.WriteLine("Main thread...");

    Thread.Sleep(2000);

    thread.Abort();

    Console.WriteLine
        ("Main thread finished");
}
```

```
static void Operation(object state)
{
    try
    {
        Console.WriteLine
            ($"Operation in state {state}");
        Thread.Sleep(1000);
        Console.WriteLine
            ("Operation finished");
    }
    catch (ThreadAbortException)
    {
        Console.WriteLine
            ("Operation was aborted");
    }
}
```

Foreground & Background

- При завершении активных foreground потоков CLR принудительно завершает все запущенные фоновые потоки
  - немедленно, без исключений
- *Foreground потоки могут быть источником багов*
  - могут мешать завершению приложения

Foreground & Background

```
public static void Main()
{
    Thread t = new Thread(Worker);
    t.IsBackground = true;
    t.Start();
    Console.WriteLine("Returning from Main");
}

private static void Worker()
{
    Thread.Sleep(10000);
    Console.WriteLine("Returning from Worker");
}
```

Планирование и приоритеты

- ОС с вытесняющей многозадачностью должна иметь алгоритм выбора порядка и продолжительности исполнения потоков
- В Windows реализуется с помощью относительных приоритетов процессов и потоков

Приоритеты потоков

Относительный приоритет потока	Класс приоритета процесса					
	Idle	Below Normal	Normal	Above Normal	High	Realtime
Time-Critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above Normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below Normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

Установка приоритета потока

- У экземпляров класса Thread можно изменять относительный приоритет при помощи свойства Priority
- Потоки с низким приоритетом хорошо использовать для длительных вычислений
- Потоки с высоким приоритетом большую часть времени находятся в режиме ожидания (Windows Explorer, мгновенная реакция на действия пользователя)

Локальное и разделяемое состояние

- У каждого потока свой стек, так что локальные переменные хранятся отдельно
- Потоки могут разделять ресурс, если имеют общую ссылку на один и тот же объект

Разделяемое состояние

```
class ThreadTest
{
    static bool _done;
    static void Main()
    {
        new Thread(Go).Start();
        Go();
    }
    static void Go()
    {
        if (!_done) { _done = true; Console.WriteLine("Done"); }
    }
}
```

Разделяемые данные – к беде

- Использование разделяемых данных (изменяемых) может приводить к ошибкам
  - даже инкремент x++ приводит к ошибкам
- Для обеспечения потокобезопасности, нужно синхронизировать совместный доступ потоков к ресурсу

Потокобезопасность

- Код **потокобезопасен** (thread-safe), если он *корректно функционирует при одновременном использовании его в нескольких потоках*
- Синхронизация – один из самых грубых способов безопасности
  - снижает производительность

Исключительная блокировка

- Простой инструмент достижения потокобезопасности – lock
- Блокируется объект ссылочного типа
- Выполнение потока блокируется в ожидании снятия блокировки, дождавшись – продолжает работу

Исключительная блокировка

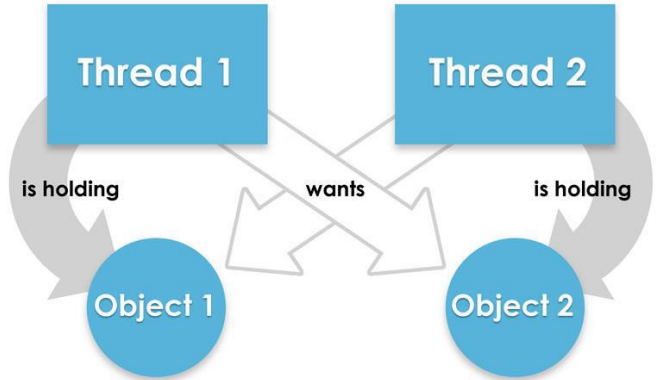
```
class ThreadTest
{
    static bool _done;
    static readonly object _locker = new object();
    static void Main()
    {
        new Thread(Go).Start();
        Go();
    }
    static void Go()
    {
        lock(_locker)
          if (!_done) { _done = true; Console.WriteLine("Done"); }
     }
}
```

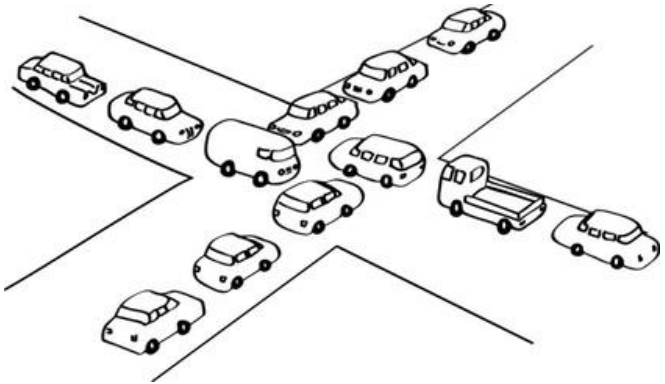
User mode синхронизация

- Конструкции синхронизации разделяемых данных
  - **Volatile** – препятствуют оптимизации компилятора и перестановке инструкций, тем самым заставляют всегда перезагружать значение из памяти (а не взятие из кэша процессора).
    - модификатор `volatile` у **полей примитивных и ссылочных типов** (а также указателей)
  - **Interlocked** –для атомарного доступа (монопольного захвата) для записи и чтения **(int, long)**
    - статические методы класса Interlocked: Increment, Decrement, Add, CompareAdd, Exchange

Синхронизация может не спасти

- **DEADLOCK**
  - Исключительное владение ресурсом
  - При этом ожидание захвата еще одного ресурса
  - Ресурс может освободиться только захватчиком
  - Циклическое ожидание





Выводы по потокам

- *Поток – дорогой ресурс*, нужно использовать аккуратно
- *У потоков можно менять тип* foreground и background
- *У потоков можно менять приоритет*
- *Потоки можно завершать и синхронизировать*

Пул потоков

- Общий для CLR набор готовых для использования потоков
  - потоков мало (сколько и процессоров), при необходимости создаются новые
  - при простое – уничтожаются
  - операции встают в очередь пула и исполняются на доступных потоках из пула

Добавление операции в очередь

- Метод `ThreadPool.QueueUserWorkItem`

```
public static void Main()
{
    Console.WriteLine("Main: queuing an asynchronous operation");
    ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);
    Console.WriteLine("Main thread: Doing other work here...");
    Thread.Sleep(10000); // Имитация другой работы (10 секунд)
}
private static void ComputeBoundOp(Object state)
{
    Console.WriteLine("In ComputeBoundOp: state={0}", state);
    Thread.Sleep(1000);
}
```

Паттерн для отмены операций

- Для отмены операций из пула есть стандартный паттерн (нельзя завершить как поток с помощью Abort)
- Используется объект состояния для скоординированной отмены CancellationTokenSource, передающий структуры CancellationToken операциям

Скоординированная отмена

```
public static void Main()
{
    CancellationTokenSource cts = new CancellationTokenSource();
    // Передаем операции CancellationToken и число
    ThreadPool.QueueUserWorkItem(o => Count(cts.Token, 1000));
    Console.WriteLine("Press <Enter> to cancel the operation.");
    Console.ReadLine();
    cts.Cancel(); // Если метод Count уже вернул управления,
                  // Cancel не оказывает никакого эффекта
                  // Cancel немедленно возвращает управление
    Console.ReadLine();
}
```

Скоординированная отмена

```
private static void Count(CancellationToken token, Int32 countTo)
{
    for (int count = 0; count < countTo; count++)
    {
        if (token.IsCancellationRequested)
        {
            Console.WriteLine("Count is cancelled");
            break; // Выход их цикла для остановки операции
        }
        Console.WriteLine(count);
        Thread.Sleep(200); // Для демонстрационных целей просто ждем
    }
            Console.WriteLine("Count is done");
}
```

Асинхронные операции

- **Асинхронность** – *форма одновременного выполнения*, при которой вызванная операция продолжает работу в фоне, не требуя от вызвавшего её кода ожидания завершения.
  По завершению, асинхронная операция продолжает работу операции (Callback)

Asynchronous Programming Model

- Вызов BeginInvoke для выполнения в контексте вторичного потока
- Вызов EndInvoke для приостановки первичного потока (Join у Thread)
- IAsyncResult для получения результата
- См ulearn и itvdn

Асинхронность

- CPU bound операции есть смысл выполнять синхронно (в том же потоке)
- I/O bound операции вместо ожидания или блокирования вызывающего потока хорошо выполнять асинхронно
- При получении команды, I/O bound операция вызывает Callback метод, позволяя обработать запрос

Выполнение I/O bound операций

- I/O bound операции, выполняемые в отдельных потоках
  - расходуют ресурсы
  - простаивая в ожидании большую часть времени
- Нужно уменьшать количество потоков, сохраняя возможности асинхронного выполнения операций

Обратный вызов при отмене

- Можно у `CancellationTokenSource` зарегистрировать несколько callback методов, которые будут вызываться в случае отмены операции (Cancel)

```
var cts = new CancellationTokenSource();
cts.Token.Register(() => Console.WriteLine("Canceled 1"));
cts.Token.Register(() => Console.WriteLine("Canceled 2"));
```

Выводы по пулу потоков

- Автоматически создает и уничтожает потоки при необходимости
- Лучше использовать вместо Thread
- Меньше накладных расходов
- Сложнее управлять
  - отменить операцию
  - узнать статус операции
  - получить результат

Задания

- Класс Task из System.Threading.Tasks
- Инструмент, упрощающий выполнение вычислительных операций в пуле потоков
- Позволяют узнать о завершении, получить возвращаемое значение вызвать Callback методы по завершению
  - Реализуют IAsyncResult

Завершение и результат

```
// Создание задания Task (оно пока не выполняется)
Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000000);

// Можно начать выполнение задания через некоторое время
t.Start();
// Можно ожидать завершения задания в явном виде

t.Wait(); // ПРИМЕЧАНИЕ. Существует перегруженная версия,
          // принимающая тайм-аут/CancellationToken

// Получение результата (свойство Result вызывает метод Wait)
Console.WriteLine("The Sum is: " + t.Result); // Значение Int32
```

Отмена задания

```
private static int Sum(CancellationToken ct, int n)
{
    int sum;
    for (sum = 0; n > 0; n--)
    {
        // Следующая строка приводит к OperationCanceledException
        // при вызове метода Cancel для CancellationTokenSource,
        // на который ссылается маркер
        ct.ThrowIfCancellationRequested();
        checked { sum += n; } // при больших n появляется
                              // исключение System.OverflowException
     }
     return sum;
}
```

Отмена задания

```
CancellationTokenSource cts = new CancellationTokenSource();
Task<int> t = new Task<int>(() => Sum(cts.Token, 10000), cts.Token);
t.Start();
cts.Cancel();// Это асинхронный запрос, задача уже может быть завершена
try
{
    // В случае отмены задания Result генерирует AggregateException
    Console.WriteLine("The sum is: " + t.Result); // Значение Int32
}
catch (AggregateException x)
{
    x.Handle(e => e is OperationCanceledException);
    // Строка выполняется, если все исключения уже обработаны
    Console.WriteLine("Sum was canceled");
}
```

Запуск нового задания по завершению

- Вместо ожидания лучше указать Callback метод, вызывающийся по завершению операции

```
// Создание объекта Task с отложенным запуском
Task<Int32> t = Task.Run(() => Sum(CancellationToken.None, 10000));
// Метод ContinueWith возвращает объект Task, но обычно
// он не используется
Task cwt = t.ContinueWith
           (task => Console.WriteLine("The sum is: " + task.Result));
```

Продолжение с условиями

```
// Создание и запуск задания с продолжением
Task<Int32> t = Task.Run(() => Sum(10000));
// Метод ContinueWith возвращает объект Task, но обычно
// он не используется
t.ContinueWith(task => Console.WriteLine("The sum is: "+task.Result),
    TaskContinuationOptions.OnlyOnRanToCompletion);
t.ContinueWith(task => Console.WriteLine("Sum threw: "+task.Exception),
    TaskContinuationOptions.OnlyOnFaulted);
t.ContinueWith(task => Console.WriteLine("Sum was canceled"),
    TaskContinuationOptions.OnlyOnCanceled);
```

Продолжение со ждуном

- У задачи Task можно получить объект TaskAwaiter, ожидающий завершения задачи и уведомляющий методы по завершению (OnCompleted) и имеющий доступ к результату (GetResult)
  - похож на итератор (это важно)

Продолжение со ждуном

```
Task<int> primeNumberTask = Task.Run(() =>
    Enumerable.Range(2, 3000000).Count(n =>
    Enumerable.Range(2, (int)Math.Sqrt(n)-1)
              .All(i => n % i > 0)));

var awaiter = primeNumberTask.GetAwaiter();
awaiter.OnCompleted(() =>
{
    int result = awaiter.GetResult();
    Console.WriteLine(result);
});
```

Вложенные задания

```
Task<Int32[]> parent = new Task<Int32[]>(() => {
var results = new Int32[2]; // Создание массива для результатов
                            // Создание и запуск 3 дочерних заданий
new Task(() => results[0] = Sum(10000),
  TaskCreationOptions.AttachedToParent).Start();
new Task(() => results[1] = Sum(20000),
  TaskCreationOptions.AttachedToParent).Start();
return results;
});

var cwt = parent.ContinueWith(
parentTask => Array.ForEach(parentTask.Result, Console.WriteLine));
parent.Start();
```

Комбинаторы All, Any

- После завершения всех или одного из дочерних заданий можно передать управление другому заданию

```
tf.ContinueWhenAll(
    childTasks,
    completedTasks => completedTasks.Where(
    t => !t.IsFaulted && !t.IsCanceled).Max(t => t.Result),
    CancellationToken.None)
.ContinueWith(t => Console.WriteLine("The maximum is: " + t.Result),
TaskContinuationOptions.ExecuteSynchronously);
});
```

Фабрики заданий

```
var cts = new CancellationTokenSource();
var tf = new TaskFactory<Int32>(cts.Token,
TaskCreationOptions.AttachedToParent,
TaskContinuationOptions.ExecuteSynchronously,
TaskScheduler.Default);
// Задание создает и запускает 3 дочерних задания
var childTasks = new[] {
tf.StartNew(() => Sum(cts.Token, 10000)),
tf.StartNew(() => Sum(cts.Token, 20000)),
tf.StartNew(() => Sum(cts.Token, Int32.MaxValue)) // Исключение
// OverflowException
};
```

Задания: выводы

- Позволяют удобно работать с пулом потоков (и не только, см. планировщики заданий)
- Позволяют узнать о завершении операции и получить результат
- Больше накладных расходов, чем у постановки в очередь пула потоков
- Можно продолжать работу вызовом Callback’a
- Можно создавать вложенные задания, комбинировать результат
- Можно с помощью фабрик создавать задания с одним состоянием

Класс Parallel

- Ещё сильнее упрощает работу с заданиями, но увеличивает overhead
- Содержит методы For, ForEach, Invoke

```
// Потоки из пула выполняют работу параллельно
Parallel.For(0, 1000, i => DoWork(i));

// Потоки из пула выполняют работу параллельно
Parallel.ForEach(collection, item => DoWork(item));

// Потоки из пула выполняют методы одновременно
Parallel.Invoke(
    () => Method1(),() => Method2(),() => Method3());
```

Parallel LINQ

- Преобразует IEnumerable последовательности в последовательности для параллельной обработки в пуле потоков ParallelEnumerable

Доступная асинхронность

- Асинхронные операции – ключ к созданию высокопроизводительных масштабируемых приложений, выполняющих операции на небольшом количестве потоков
- CLR предлагает доступную модель для реализации асинхронности
  - основанную на задачах Task и асинхронных методах
  - по сути, просто *синтаксический сахар*

Асинхронные методы

```
var result = await expression;
statement(s);
```

преобразуется в

```
var awaiter = expression.GetAwaiter();
awaiter.OnCompleted(() =>
{
    var result = awaiter.GetResult();
    statement(s);
});
```

Описание процесса

- Оператор await преобразует операнд в вызов для него метода GetAwaiter
- Запрашивается IsCompleted у объекта ожидания и вызывает GetResult при успехе
- Если IsCompeted == false вызывается метод продолжения OnCompleted

Количество простых чисел

```
int GetPrimesCount(int start, int count)
{
    return ParallelEnumerable.Range(start, count).Count(n =>
        Enumerable.Range(2, (int)Math.Sqrt(n) - 1)
                  .All(i => n % i > 0));
}
void DisplayPrimeCounts()
{
    for (int i = 0; i < 10; i++)
        Console.WriteLine(
            GetPrimesCount(i * 1000000 + 2, 1000000) +
            " primes between " + (i * 1000000) + " and " +
            ((i + 1) * 1000000 - 1));
    Console.WriteLine("Done!");
}
```

Асинхронный вариант

```
Task<int> GetPrimesCountAsync(int start, int count)
{
    return Task.Run(() =>
        ParallelEnumerable.Range(start, count)
        .Count(n =>
            Enumerable.Range(2, (int)Math.Sqrt(n) - 1)
        .All(i => n % i > 0)));
}
async Task DisplayPrimeCountsAsync()
{
    for (int i = 0; i < 10; i++)
        Console.WriteLine(await GetPrimesCountAsync(i * 1000000 + 2,
        1000000) + " primes between " + (i * 1000000) + " and " +
        ((i + 1) * 1000000 - 1));
Console.WriteLine("Done!");
}
```

Конечный автомат

- Асинхронные методы – конечный автомат, подобный автоматам последовательностей IEnumerable
- Применение await похоже на вызов методов расширения LINQ

Стандартные асинхронные методы

- Все классы, производные от System.IO.Stream содержат ReadAsync, WriteAsync, FlushAsync, CopyAsync
- Производные от System.Net.HttpClient содержат GetAsync, PostAsync, PutAsync,…
- System.Data.SqlClient.SqlCommand содержит ExecuteDbDataReaderAsync, ExecuteNonQueryAsync,…

Другие возможности

- Можно группировать операции в коллекцию и вызывать await с продолжением по завершению всех или хотя бы одной операции

```
List<Task<String>> requests = new List<Task<String>>(10000);
for (Int32 n = 0; n < requests.Capacity; n++)
    requests.Add(IssueClientRequestAsync("localhost","Request #"+n));
String[] responses = await Task.WhenAll(requests);
```

Асинхронные методы: выводы

- Async, await – синтаксический сахар, упрощающий реализацию асинхронного выполнения операций с помощью заданий
- Асинхронные методы – конечный автомат (на подобие автоматов последовательностей)

Остались за бортом

- **Таймеры (Timers)** для выполнения повторяющихся заданий
- **Механизмы синхронизации**
  - **События** (bool переменные под упр. ядра с сообщением об изменении потокам)
  - **Семафоры** (int переменные под упр. ядра – счетчики, блокирующие потоки)
  - **Мьютексы** (более сложные объекты для взаимно исключающей блокировки с дополнительной информацией о потоке-владельце)
- **Гибридная синхронизация**
  - **Monitor** – еще более мощный и сложный инструмент для взаимно исключающей блокировки

Коллекции для параллельного доступа

- Потокобезопасные коллекции, реализуют IProducerConsumerCollection
  - ConcurrentStack
  - ConcurrentQueue
  - ConcurrentDictionary
  - ConcurrentBag

Что почитать/посмотреть

- Рихтер – CLR via C# 4.5 (и новее)
- Альбахари – C# 7.0. Справочник. Полное описание языка (лучше в оригинале C# 7.0 in a nutshell)
- http://cdn.oreillystatic.com/oreilly/booksamplers/9781449367565_sampler.pdf
- http://download.microsoft.com/download/5/B/9/5B924336-AA5D-4903-95A0-56C6336E32C9/TAP.docx
- https://www.youtube.com/watch?v=fi_N_ghu4Ug&list=PLvItDmb0sZw-sOL6sOsEnSJ8etu7Kbgko&index=15

Вопросы?

e-mail: marchenko@it.kfu.ru