СВОБОДА ОТ БЛОКИРОВОК ИЛИ LOCK-FREE МНОГОПОТОЧНОСТЬ

КЕРБИЦКОВ ЮРИЙ

KZNDOTNET MEETUP #1

НЕБЛОКИРУЮЩАЯ СИНХРОНИЗАЦИЯ

Подход в параллельном программировании на симметрично-многопроцессорных системах, в котором принят отказ от традиционных примитивов блокировки, таких, как семафоры, мьютексы и события.

АЛГОРИТМЫ

- Obstruction Free

АЛГОРИТМЫ

- Obstruction Free
- Lock Free

АЛГОРИТМЫ

- Obstruction Free
- Lock Free
- Wait Free

КОНСТРУКЦИЯ LOCK

КОНСТРУКЦИЯ LOCK

```
private static object syncObject = new object();

0 references
private static void Foo()
{
    lock (syncObject)
    {
        // ...
    }
}
```

КОНСТРУКЦИЯ LOCK

```
private static object syncObject = new object();

0 references
private static void Foo()
{
    lock (syncObject)
    {
        // ...
    }
}
```

```
private static object syncObject = new object();

0 references
private static void Foo()
{
    bool lockTaken = false;
    try
    {
        Monitor.Enter(syncObject, ref lockTaken);
        // ...
    }
    finally
    {
        if (lockTaken) Monitor.Exit(syncObject);
    }
}
```

ЧТО НУЖНО ПОМНИТЬ ПРО MONITOR

- Реализует паттерн «условная переменная» (Conditional Variable)
- Использует мьютекс
- Большую часть времени выполняется в пользовательском режиме
- Переходит в режим ядра при длительном времени ожидания

ПЕРЕСТАНОВКИ

КОМПИЛЯТОР

- Выполнение меньшего количества инструкций
- Оптимизация использования регистров
- Уменьшение количества обращений к памяти

КОМПИЛЯТОР

```
for (int i = 0; i < n; i++)
{
    x = y + z;
    a[i] = 6 * i + x * x;
}
```

КОМПИЛЯТОР

```
for (int i = 0; i < n; i++)
{
    x = y + z;
    a[i] = 6 * i + x * x;
}
```

```
x = y + z;
b = x * x;
for (int i = 0; i < n; i++)
    a[i] = 6 * i + b;
```

ПРОЦЕССОР

Техники распараллеливания инструкций:

- Вычислительный конвейер
- Суперскалярный процессор
- Модуль предсказания переходов (прогнозирование ветвлений)

ПРОЦЕССОР

З – запись данных в память
Ч – чтение данных из памяти
В – выполнение инструкции

ПРОЦЕССОР

З – запись данных в память
Ч – чтение данных из памяти
В – выполнение инструкции

Скалярный

Цикл	0	1	2	3	4
Инструкция 1	Ч	В	З		
Инструкция 2		Ч	В	З	
Инструкция 3			Ч	В	З

ПРОЦЕССОР

З – запись данных в память
Ч – чтение данных из памяти
В – выполнение инструкции

Супер скалярный

Цикл	0	1	2	3	4
Инструкция 1	Ч	В	З		
Инструкция 2	Ч	В	З		
Инструкция 3		Ч	В	З	
Инструкция 4		Ч	В	З	
Инструкция 5			Ч	В	З
Инструкция 6			Ч	В	З

КОГЕРЕНТНОСТЬ КЭША

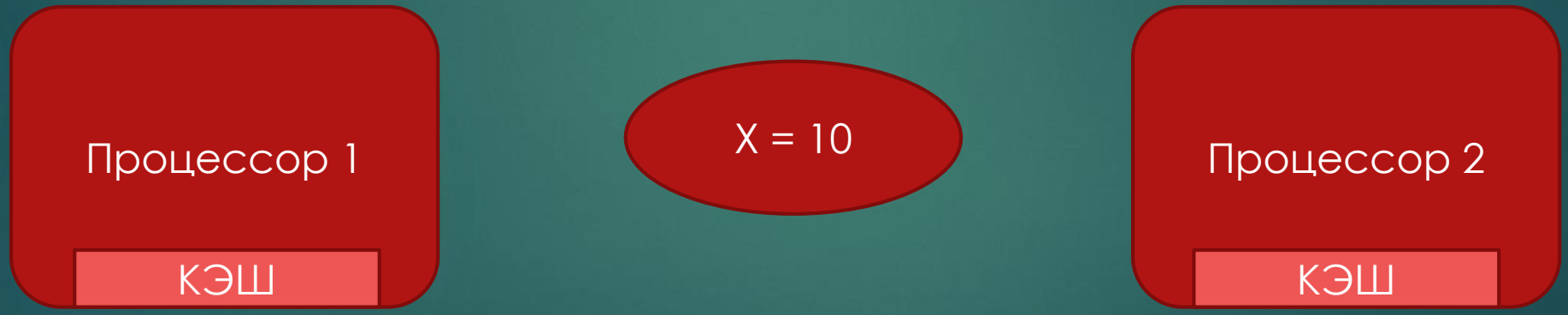
- Разные архитектуры предоставляют разные политики когерентности

КОГЕРЕНТНОСТЬ КЭША

- Разные архитектуры предоставляют разные политики когерентности

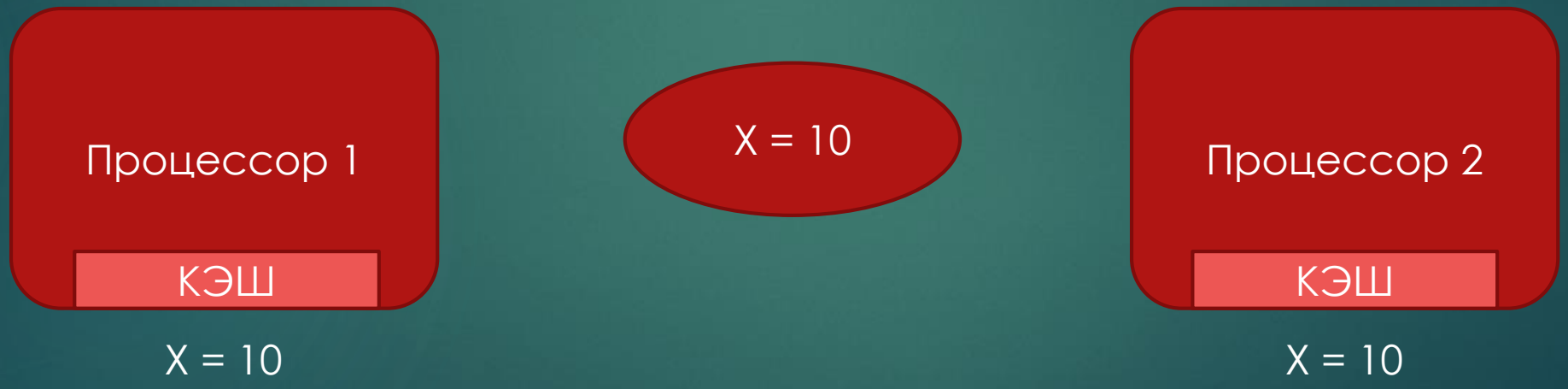
КОГЕРЕНТНОСТЬ КЭША

- Разные архитектуры предоставляют разные политики когерентности



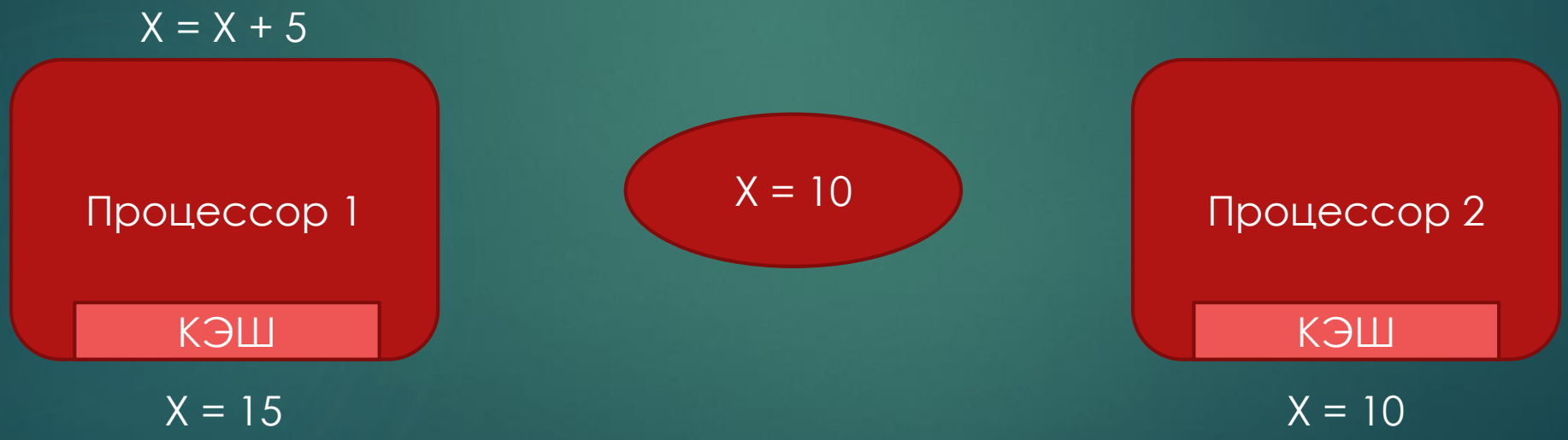
КОГЕРЕНТНОСТЬ КЭША

- Разные архитектуры предоставляют разные политики когерентности



КОГЕРЕНТНОСТЬ КЭША

- Разные архитектуры предоставляют разные политики когерентности



ЧТО В ИТОГЕ?

- Последовательность следования инструкций такая, какой мы написали в коде
- Реальная последовательность выполнения инструкций
- Возможные последовательности выполнения инструкций
- Все платформы имеют модель памяти, которая четко описывает допустимые перестановки

АТОМАРНОСТЬ

АТОМАРНОСТЬ

- Операция записи/чтения 32-х битных и менее значений всегда атомарна
- Операция записи/чтения 64-х битных значений атомарна только в 64-х битной ОС

АТОМАРНОСТЬ

- Операция записи/чтения 32-х битных и менее значений всегда атомарна
- Операция записи/чтения 64-х битных значений атомарна только в 64-х битной ОС

Атомарные типы данных

- bool, char, byte, sbyte, short, ushort, uint, int, float
- Ссылочные типы
- Перечисления, чей базовый тип: byte, sbyte, short, ushort, int, uint

EXCHANGE

```
public static int Exchange(ref int location1, int value);
```

- Прочитать значение и заменить новым
- XCHG инструкция

```
0 references
struct SpinLock
{
    private int m_taken;

    0 references
    public void Enter()
    {
        while (Interlocked.Exchange(ref m_taken, 1) != 0) /*spin*/;
    }

    0 references
    public void Exit() => m_taken = 0;
}
```

COMPARE AND EXCHANGE (CAS)

```
public static int CompareExchange(ref int location1, int value, int comparand);
```

- Прочитать значение, сравнить в компарандом. Если равны, то записать новое значение
- CMPXCHG инструкция

```
public static int CompareExchage(ref int location1, int value, int comparand)
{
    var original = location1;
    if (location1 == comparand)
        location1 = value;

    return original;
}
```

COMPARE AND EXCHANGE (CAS)

```
struct SpinLock
{
    private int m_taken;

    0 references
    public void Enter()
    {
        int mid = Thread.CurrentThread.ManagedThreadId;
        while (Interlocked.CompareExchange(ref m_taken, mid, 0) != 0) /*spin*/;
    }

    0 references
    public void Exit() => m_taken = 0;
}
```

64 бита

64 бита

- CMPXCHG8B инструкция

64 бита

- CMPXCHG8B инструкция

```
public static long Read(ref long location);
```

64 бита

- CMPXCHG8B инструкция

```
public static long Read(ref long location);
public static long CompareExchange(ref long location1, long value, long comparand);
```

64 бита

- CMPXCHG8B инструкция

```
public static long Read(ref long location);
```

```
public static long CompareExchange(ref long location1, long value, long comparand);
```

```
static void AtomicWrite(ref long location, long value)
{
    if (IntPtr.Size == 4) Interlocked.Exchange(ref location, value);
    else location = value;
}

0 references
static long AtomicRead(ref long location)
{
    if (IntPtr.Size == 4) return Interlocked.CompareExchange(ref location, 0L, 0L);
    else return location;
}
```

БОЛЬШЕ АТОМАРНОСТИ!

```
public static long Add(ref long location1, long value);
public static int Add(ref int location1, int value);
```

```
public static int Decrement(ref int location);
public static long Decrement(ref long location);
```

```
public static int Increment(ref int location);
public static long Increment(ref long location);
```

СОГЛАСОВАННОСТЬ ПАМЯТИ

МОДЕЛИ ПАМЯТИ

- Слабая – разрешает переставлять все операции чтения/записи
- Сильная – не разрешает никаких перестановок. Код, который выполняется = исходному коду, который написан разработчиком

МОДЕЛИ ПАМЯТИ

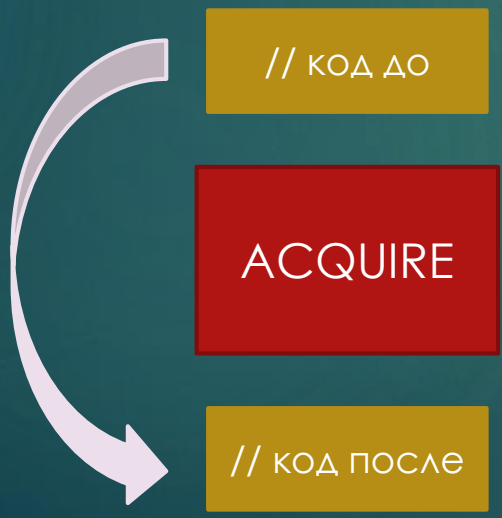
	X86	AMD64	ARMv7
Чтение после Чтения	-	-	+
Чтение после Записи	-	-	+
Запись после Записи	-	-	+
Запись после Чтения	+	+	+

БАРЬЕРЫ ПАМЯТИ. ПРОЦЕССОР

- Полный барьер – никакие перестановки операций чтения/записи не могут быть сделаны через этот барьер (MFENCE инструкция)
- Барьер операции записи (SFENCE инструкция) – запрещает перестановки операций записи через себя
- Барьер операции чтения (LFENCE) – запрещает перестановки операций чтения через себя

БАРЬЕРЫ ПАМЯТИ. КОМПИЛЯТОР

- Acquire – запрещает операции чтения/записи переносить и ставить перед собой
- Release – запрещает операции чтения/записи переносить и ставить после себя



БАРЬЕРЫ ПАМЯТИ

- Все Interlocked методы устанавливают полный барьер памяти

БАРЬЕРЫ ПАМЯТИ

- Все Interlocked методы устанавливают полный барьер памяти

```
Thread.MemoryBarrier();
```
- полный барьер

БАРЬЕРЫ ПАМЯТИ

- Все Interlocked методы устанавливают полный барьер памяти

```
Thread.MemoryBarrier();
```

- полный барьер

```
var x_copy = Thread.VolatileRead(ref x);
```

```
static volatile int y;
var x_copy = y;
```

Acquire барьер

БАРЬЕРЫ ПАМЯТИ

- Все Interlocked методы устанавливают полный барьер памяти

```
Thread.MemoryBarrier();
```
- полный барьер

```
var x_copy = Thread.VolatileRead(ref x);
```

```
static volatile int y; var x_copy = y;
```

Acquire барьер

```
Thread.VolatileWrite(ref x, 50);
```

```
static volatile int y; y = 50;
```

Release барьер

ПРИМЕР. ПОТОКОБЕЗОПАСНЫЙ СТЕК

ЭЛЕМЕНТ СТЕКА

```
8 references
internal class Node<T>
{
    2 references
    public Node(T value, Node<T> next)
    {
        Value = value;
        Next = next;
    }

    3 references
    public T Value { get; set; }
    3 references
    public Node<T> Next { get; set; }
}
```

РЕАЛИЗАЦИЯ МЕТОДА PUSH

```
3 references
public void Push(T value)
{
    lock (syncObject)
    {
        head = new Node<T>(value, head);
    }
}
```

РЕАЛИЗАЦИЯ МЕТОДА PUSH

```
3 references
public void Push(T value)
{
    lock (syncObject)
    {
        head = new Node<T>(value, head);
    }
}
```

```
3 references
public void Push(T value)
{
    Node<T> node, oldHead;
    do
    {
        oldHead = head;
        node = new Node<T>(value, oldHead);
    } while (Interlocked.CompareExchange(ref head, node, oldHead) != oldHead);
}
```

РЕАЛИЗАЦИЯ МЕТОДА TRYPOP

```
3 references
public bool TryPop(out T data)
{
    lock (syncObject)
    {
        if (head == null)
        {
            data = default(T);
            return false;
        }

        data = head.Value;
        head = head.Next;
        return true;
    }
}
```

РЕАЛИЗАЦИЯ МЕТОДА TRYPOP

```
3 references
public bool TryPop(out T data)
{
    lock (syncObject)
    {
        if (head == null)
        {
            data = default(T);
            return false;
        }

        data = head.Value;
        head = head.Next;
        return true;
    }
}
```

```
3 references
public bool TryPop(out T data)
{
    var oldHead = head;
    while (oldHead != null)
    {
        if (oldHead == Interlocked.CompareExchange(ref head,
            oldHead.Next, oldHead))
        {
            data = oldHead.Value;
            return true;
        }

        oldHead = head;
    }

    data = default(T);
    return false;
}
```

БЕНЧМАРКИНГ

```
public void TestLock(IStack<int> stack)
{
    var iterations = 10;
    var threads = 100;

    var pushTasks = Enumerable.Range(0, threads).Select(_ => Task.Run(() =>
    {
        for (int i = 0; i < iterations; i++)
            stack.Push(i);
    }));
    Task.WaitAll(pushTasks.ToArray());

    var popTasks = Enumerable.Range(0, threads).Select(_ => Task.Run(() =>
    {
        for (int i = 0; i < iterations; i++)
            stack.TryPop(out int _);
    }));
    Task.WaitAll(popTasks.ToArray());
}
```

БЕНЧМАРКИНГ

```
[Benchmark(OperationsPerInvoke = 10)]
0 references
public void TestLock() => TestLock(new LockStack<int>());

[Benchmark(OperationsPerInvoke = 10)]
0 references
public void TestFree() => TestLock(new FreeStack<int>());
```

БЕНЧМАРКИНГ

```
BenchmarkDotNet=v0.10.13, OS=Windows 10 Redstone 3 [1709, Fall Creators Update] (10.0.16299.192)
Intel Core i3-2310M CPU 2.10GHz (Sandy Bridge), 1 CPU, 4 logical cores and 2 physical cores
Frequency=2046133 Hz, Resolution=488.7268 ns, Timer=TSC
  [Host] : .NET Framework 4.6.2 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.7.2600.0
  Clr    : .NET Framework 4.6.2 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.7.2600.0

Job=Clr  Runtime=Clr
```

Method	Mean	Error	StdDev
TestLock	36.43 us	0.5913 us	0.5531 us
TestFree	22.38 us	0.4469 us	0.8395 us

ЗАКЛЮЧЕНИЕ

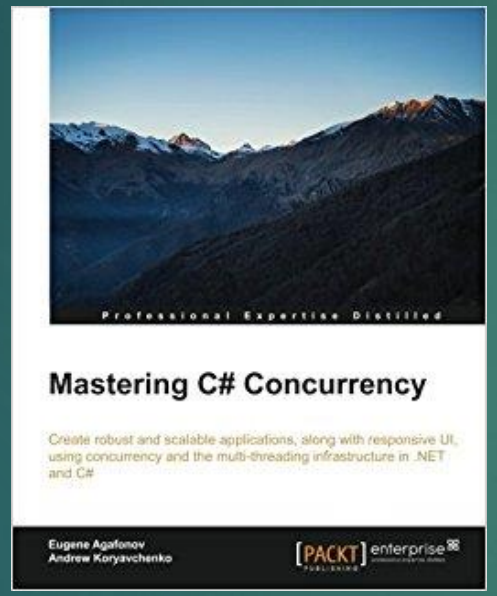
ДОСТОИНСТВА

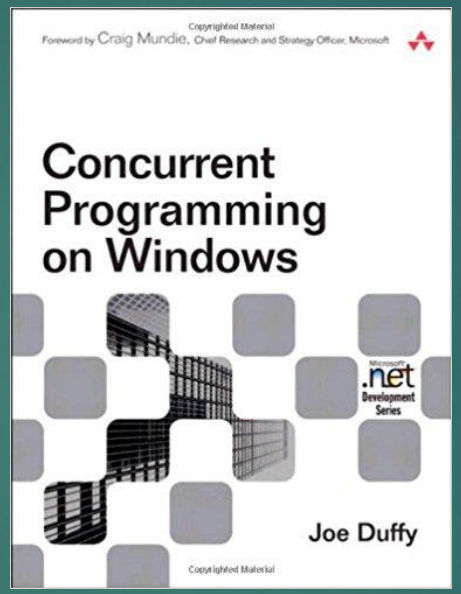
- Производительность

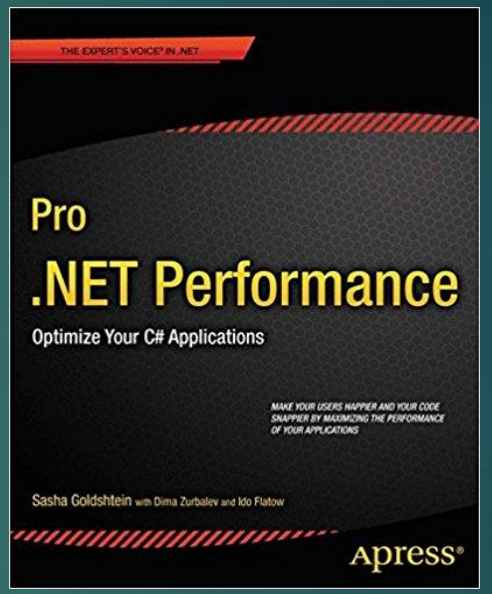
НЕДОСТАТКИ

- Требуется высокая квалификация и обширные знания для корректной реализации алгоритмов
- При длительных периодах ожидания увеличивается утилизация процессора и возрастает нагрузка на кэш для поддержания когерентности
- Код становится более запутанным и сложным

ЧТО ПОЧИТАТЬ







- Спецификации языка и среды – ECMA 334, ECMA 335

ВОПРОСЫ?

Кербицков Юрий

kirbex@mail.ru

https://github.com/kirbex

https://www.nuget.org/profiles/kirbex

https://stackoverflow.com/users/2753469/yury-kerbitskov