t.me/epeshkblog

Асинхронность: не только async/await

Евгений Пешков

@epeshk

О чём будем говорить?

- Зачем нужна асинхронность
- Особенности реализации async/await
- Подходы из других платформ
- Эксперименты в .NET рантайме
  - Green threads
  - async2

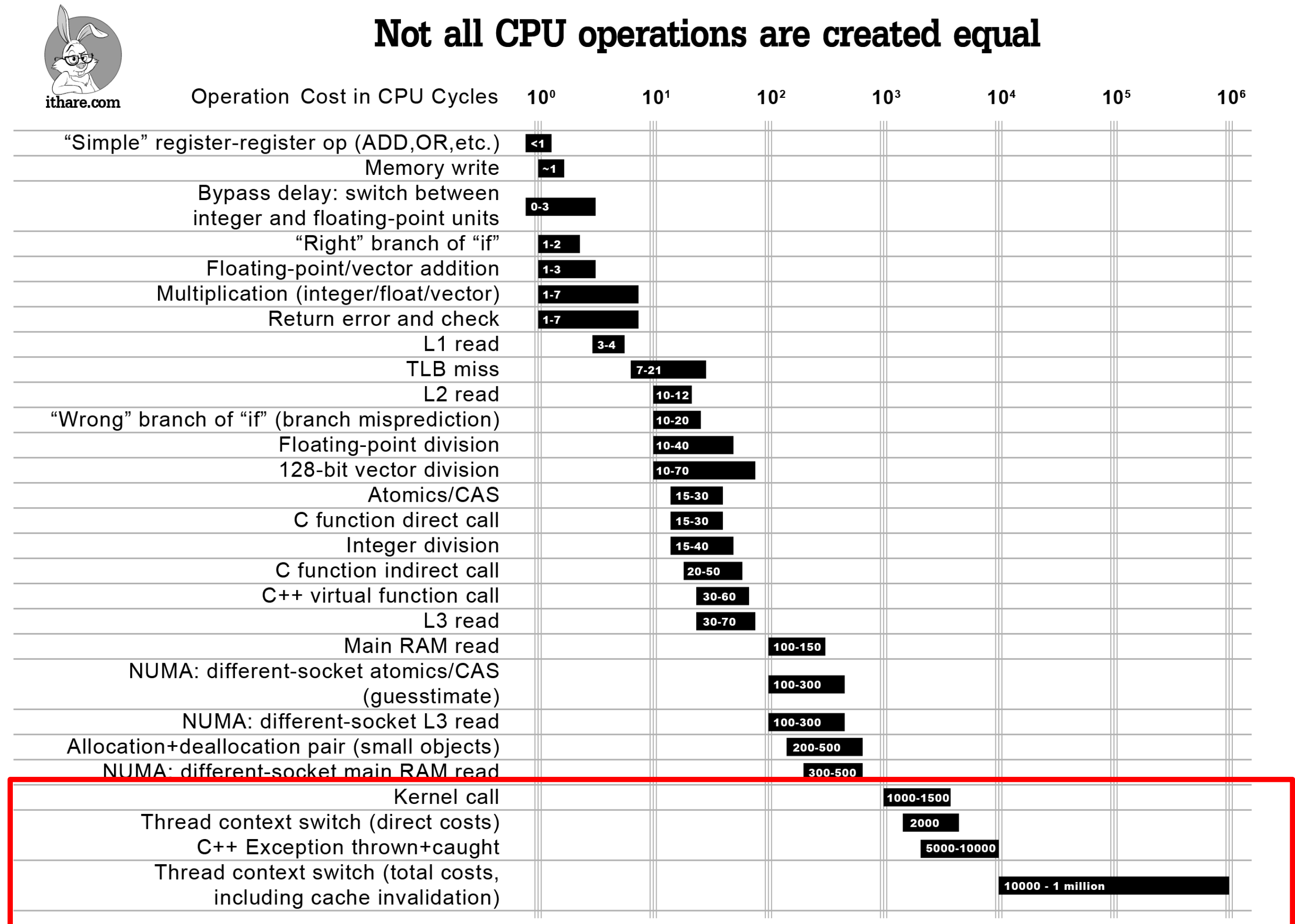
t.me/epeshkblog

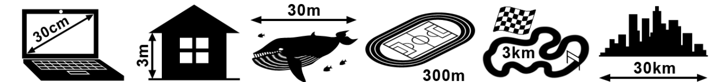
Зачем нужна асинхронность

- Создание потока – медленно, лучше использовать потоки из Thread Pool
- Чтобы не выделять память под стек
- Чтобы не зависал UI
- Чтобы процессор не простаивал при ожидании IO

Зачем нужна асинхронность

- Асинхронность – перенос scheduling'а задач с уровня ядра ОС на уровень приложения, в user mode
- Позволяет сэкономить на медленных context switch на уровне CPU





Async/await: история

- Появился в C# 5 (2012)
- Основной подход: асинхронный код должен читаться как синхронный
- Позволяет избавиться от callback hell

```
File.ReadAllTextAsync("data.txt")
    .ContinueWith(x => Console.WriteLine(x.Result));
```

- Синтаксический сахар – языковая фича, не требующая поддержки рантаймом

t.me/epeshkblog

roslyn AsyncRewriter

```
static async Task<int> AsyncCaller()
{
    var a = await AsyncMethod();
    var b = await AsyncMethod();
    return a + b;
}
```

```
[AsyncStateMachine(typeof (<AsyncCaller>d__1))]
static Task<int> AsyncCaller()
{
  <AsyncCaller>d__1 stateMachine;
  stateMachine.builder = AsyncTaskMethodBuilder<int>.Create();
  stateMachine.state = -1;
  stateMachine.builder.Start(ref stateMachine);
  return stateMachine.builder.Task;
}

struct <AsyncCaller>d__1 : IAsyncStateMachine
{
  public int state;
  public AsyncTaskMethodBuilder<int> builder;
  private int __2;
  private TaskAwaiter<int> __1;

  void IAsyncStateMachine.MoveNext() {
    switch (state) { … }
  }
  void SetStateMachine(IAsyncStateMachine stateMachine);
}

// dotPeek: Low-level C#
```

Async/await: проблемы

Дизайн-проблемы: недостатки самого синтаксиса async/await

Технические проблемы: особенности реализации async/await в языке программирования

t.me/epeshkblog

Красно-синие функции (2015)

- У каждой функции есть цвет (синий или красный)
- Правила вызова функции зависят от цвета
- Красные функции можно вызвать только из красных функций
- Без красных функций код выглядит проще
- *Sadistic language designers* заставляют использовать красные функции

- Красные функции – асинхронные

https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/ (Bob Nystrom)

Красно-синие функции в C#

- await можно использовать только в асинхронных функциях
- Синхронные функции можно вызывать везде
- Дублирование кода: `Read` и `ReadAsync`

- Блокирующее ожидание для вызова красного из синего:
  - `.Result`
  - `.GetAwaiter().GetResult()`

- Возможность блокировки – простор для багов

t.me/epeshkblog

Вызов асинхронного метода без await

```
var periodicTimer = new PeriodicTimer(TimeSpan.FromSeconds(1));

while (true)
{
    RebuildCacheAsync();
    await periodicTimer.WaitForNextTickAsync();
}
```

К чему приведёт ошибка в этом коде?

Вызов асинхронного метода без await

```
var periodicTimer = new PeriodicTimer(TimeSpan.FromSeconds(1));

while (true)
{
    RebuildCacheAsync();
    await periodicTimer.WaitForNextTickAsync();
}
```

- Избыточный параллелизм
- Сломанная логика (а если cache single writer?)
- Возможно пропустить исключение (fire and forget)

```
TaskScheduler.UnobservedTaskException += (s, e) => LogError(e.Exception);
```

t.me/epeshkblog

Вызов синхронного кода на ThreadPool

- Thread Pool starvation: нехватка потоков в пуле для разбора очереди
- Полная блокировка программы: созданные потоки сразу же блокируются синхронным кодом

- Иногда вызов синхронного кода предпочтительнее: файловый IO с SSD

Синхронное ожидание async-метода

```
async Task MyAsyncMethod() {
  var data = LibraryMethod();
  await ProcessDataAsync(data);
}

string LibraryMethod()
  => FetchDataAsync().GetAwaiter().GetResult();

Task<string> FetchDataAsync();
```

Фейковая асинхронность

```
app.MapGet("/api", async ctx =>
{
  // логируем host name, откуда пришел запрос
  var ip = ctx.Connection.RemoteIpAddress;
  var hostName = (await Dns.GetHostEntryAsync(ip)).HostName;
  LogRequest(clientHostName);
  ...
})
```

Фейковая асинхронность

```
app.MapGet("/api", async ctx =>
{
  // логируем host name, откуда пришел запрос
  var ip = ctx.Connection.RemoteIpAddress;
  var hostName = (await Dns.GetHostEntryAsync(ip)).HostName;
  LogRequest(clientHostName);
  ...
})
```

- Reverse DNS lookup на *nix работает синхронно
- Результат – каждый входящий запрос блокирует поток на время DNS lookup

t.me/epeshkblog

Аллокации

Состояние асинхронной state machine хранится в куче:

- `AsyncStateMachineBox<TStateMachine>`
- `Task object`

t.me/epeshkblog

Аллокации: ValueTask

ValueTask/ValueTask<T> позволяют избежать аллокаций

- Структура – если метод завершился синхронно, ничего не аллоцируется
- Можно переиспользовать IValueTaskSource для асинхронного ожидания
- За отсутствие аллокаций отвечает *вызываемый* метод

```
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder))]
async ValueTask AsyncMethod() => await Task.Yield();
```

Медленные исключения

```
void SyncMethod(int depth = 0)
{
  if (depth == ThrowDepth)
    throw new Exception();

  SyncMethod(depth + 1);
}

async Task AsyncMethod(int depth = 0)
{
  if (depth == ThrowDepth)
    throw new Exception();

  await AsyncMethod(depth + 1);
}
```

Method	ThrowDepth	Mean
Sync	1	5 us
Async	1	14 us
Sync	10	11 us
Async	10	65 us
Sync	100	75 us
Async	100	650 us
Sync	1000	705 us
Async	1000	22867 us
Sync	10000	7791 us
Async	10000	NA

Медленные исключения

38,8% Async • 18 930 ms • Async()
  21,0% HandleNonSuccessAndDebuggerNotification • 10 222 ms • HandleNonSuccessAndDebuggerNotification()
    20,5% ThrowForNonSuccess • 10 006 ms • System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess()
      20,0% Throw • 9 747 ms • System.Runtime.ExceptionServices.**ExceptionDispatchInfo.Throw**()

t.me/epeshkblog

Восстановление stack trace

Реальный стектрейс потока:

- `PortableThreadPool+WorkerThread.WorkerThreadStart()`
  - `LowLevelLifoSemaphore.Wait(Int32, Boolean)`
    - `ThreadPoolWorkQueue.Dispatch()`
      - `IThreadPoolWorkItem.Execute()`
        - ``AsyncTaskMethodBuilder`1+AsyncStateMachineBox`1.MoveNext(Thread)``
          - `ExecutionContext.RunFromThreadPoolDispatchLoop(...)`
            - `Demo+<CAsync>d__7.MoveNext()`

Асинхронный стектрейс:

- `await AAsync()`
  - `await BAsync()`
    - `await CAsync()`

Сложности отладки

- Для обычных потоков легко посмотреть stack trace
- Для Task требуется обойти кучу в поиске state machines
  - `dotnet dump: dumpasync`

t.me/epeshkblog

Решения конкурентов (stackful)

- Go: goroutines
- Java: virtual (green) threads, project loom

- Асинхронность – перенос scheduling'а задач с уровня ядра ОС на уровень приложения, в user mode
- Scheduling ~ потоки

- A *goroutine* is a lightweight thread of execution
- *Virtual (green) threads* are lightweight threads that dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications
- Preserving the thread-per-request style with virtual (green) threads

Stackful/stackless

- Stackless (C#, Kotlin, Python, JavaScript): Не имеют собственного стека. В куче сохраняется минимальное состояние, необходимое для возобновления выполнения (AsyncStateMachine)

- Stackful (Go, Java): Каждая корутина имеет свой собственный стек. Стек хранится в памяти, даже в состоянии ожидания
  - Для виртуального потока используется не стек нативного потока, а кастомная реализация растущего стека
  - Внутри – тот же Thread Pool, когда virtual thread блокируется – поток пула начинает выполнять следующий готовый virtual thread

Virtual thread: два потока

- Virtual thread – логический поток исполнения
- Platform (carrier) thread – поток из пула, выполняющий код разных virtual threads

- Стек virtual thread – просто область памяти, не привязанная жестко к platform thread
- Для экономии памяти стек virtual thread делается растущим
  - Go: 2 KB
  - Kotlin: 1 KB

Virtual thread example: echo server

```
Socket clientSocket = serverSocket.accept();

Thread.ofVirtual().start(() -> {
  try (
    PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
    ) {
      String inputLine;
      while ((inputLine = in.readLine()) != null) {
        System.out.println(inputLine);
        out.println(inputLine);
      }

    } catch (IOException e) {
      e.printStackTrace();
    }
});
```

- **Thread.*ofVirtual*()**
- **Thread.*ofPlatform*()**

t.me/epeshkblog

Virtual thread: решаемые проблемы

Virtual threads лишены недостатков дизайна async/await:

- Красно-синие функции
- Дублирование кода
- Простор для багов
  - Код работает синхронно или асинхронно в зависимости от потока выполнения, а не способа вызова
  - Однако, абсолютно неявно: невозможно вызвать синхронную версию при необходимости

t.me/epeshkblog

Эксперименты в .NET

- .NET 8: Green threads
  - Реализация аналогичных Java virtual threads
- .NET 9: async2
  - Перенос async/await с уровня IL кода в рантайм

t.me/epeshkblog

Green threads

github.com/dotnet/runtimelab/tree/feature/green-threads

```
struct GreenThreadData
{
    StackRange osStackRange;
    uint8_t* osStackCurrent;
    uint8_t* greenThreadStackCurrent;
    Frame* pFrameInGreenThread;
    Frame* pFrameInOSThread;
    GreenThreadStackList *pStackListCurrent;
    bool inGreenThread;
    bool transitionedToOSThreadOnGreenThread;
};
```

t.me/epeshkblog

Green thread API changes

- `Thread.IsGreenThread`
  - Проверяет, запущен ли код на green thread
- `Task.RunAsGreenThread(Action)`
  - Запускает синхронный Action как асинхронный на green thread
- `Task.Wait() / .GetAwaiter().GetResult()`
  - При вызове в green thread выполняет неблокирующее ожидание

t.me/epeshkblog

Green thread stub

```
class Socket {
  public int Send_Old(byte[] buffer) { ... }

  public ValueTask<int> SendAsync(byte[] buffer) { ... }

  public int Send(byte[] buffer) {
    if (Thread.IsGreenThread)
    {
      ValueTask<int> vt = SendAsync(buffer);
      return vt.IsCompleted ?
        vt.GetAwaiter().GetResult() :
        vt.AsTask().GetAwaiter().GetResult();
    }

    return Send_Old(buffer);
  }
}
```

- TaskAwaiter.GetResult();
  - Task.SpinThenBlockingWait();
    - Task.YieldGreenThread();
      - greenthreads.cpp:GreenThread_Yield();

t.me/epeshkblog

Green thread stub for Span<T>

```
class Socket {
  public int Send_Old(Span<byte> buffer) { ... }

  public ValueTask<int> SendAsync(Memory<byte> buffer) { ... }

  public int Send(Span<byte> buffer) {
    if (Thread.IsGreenThread)
    {
      ValueTask<int> vt = SendAsync(...);
      return vt.IsCompleted ?
        vt.GetAwaiter().GetResult() :
        vt.AsTask().GetAwaiter().GetResult();
    }

    return Send_Old(buffer);
  }
}
```

t.me/epeshkblog

Green thread stub for Span<T>

```
class Socket {
  public int Receive(Span<byte> buffer) {
    if (Thread.IsGreenThread) {
      byte[] tmp = ArrayPool<byte>.Shared.Rent(buffer.Length);
      try {
        buffer.CopyTo(tmp);
        ValueTask<int> vt = ReceiveAsync(tmp);

        return vt.IsCompleted ?
          vt.GetAwaiter().GetResult() :
          vt.AsTask().GetAwaiter().GetResult();
      }
      finally {
        ArrayPool<byte>.Shared.Return(tmp);
      }
    }

    return Receive_Old(buffer);
  }
}
```

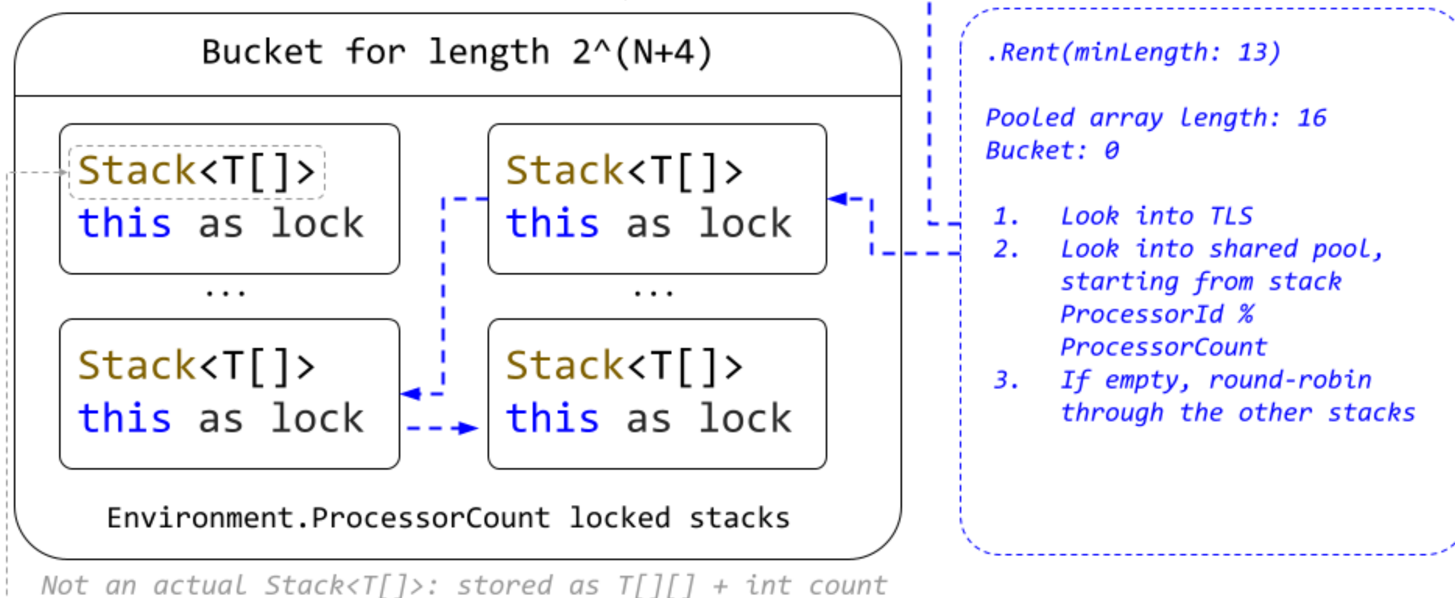
t.me/epeshkblog

Thread locals

ArrayPool<T>.Shared

[ThreadStatic] T[][]: Thread local cache
Single slot per length

PerCoreLockedStacks[]: [_16, _32, _64, ..., 1024*1024*1024]



Thread locals

- К какому потоку относится `[ThreadStatic]`? К virtual или platform?
- Если отнести `[ThreadStatic]` к green thread – подобная реализация пулинга станет бесполезной
- Подобная оптимизация в .NET используется часто – в итоге `[ThreadStatic]` отнесли к platform thread
- В Java наоборот, `ThreadLocal<T>` относится к virtual thread

Недостатки green threads

- Заражение синхронного кода stub'ами
- Проблема с методами, принимающими Span<T>
  - Для перехода в Async-версию требуется копирование (в массив из пула)
- Медленный P/Invoke (6 раз)
- По производительности оказался таким же, как async (но они и не про перформанс?!):
  - ASP.NET Core plaintext: 178K RPS (async), 162K RPS (green threads)
- Различные сложности реализации

- На этом эксперимент с green threads закончили, и перешли к следующему…

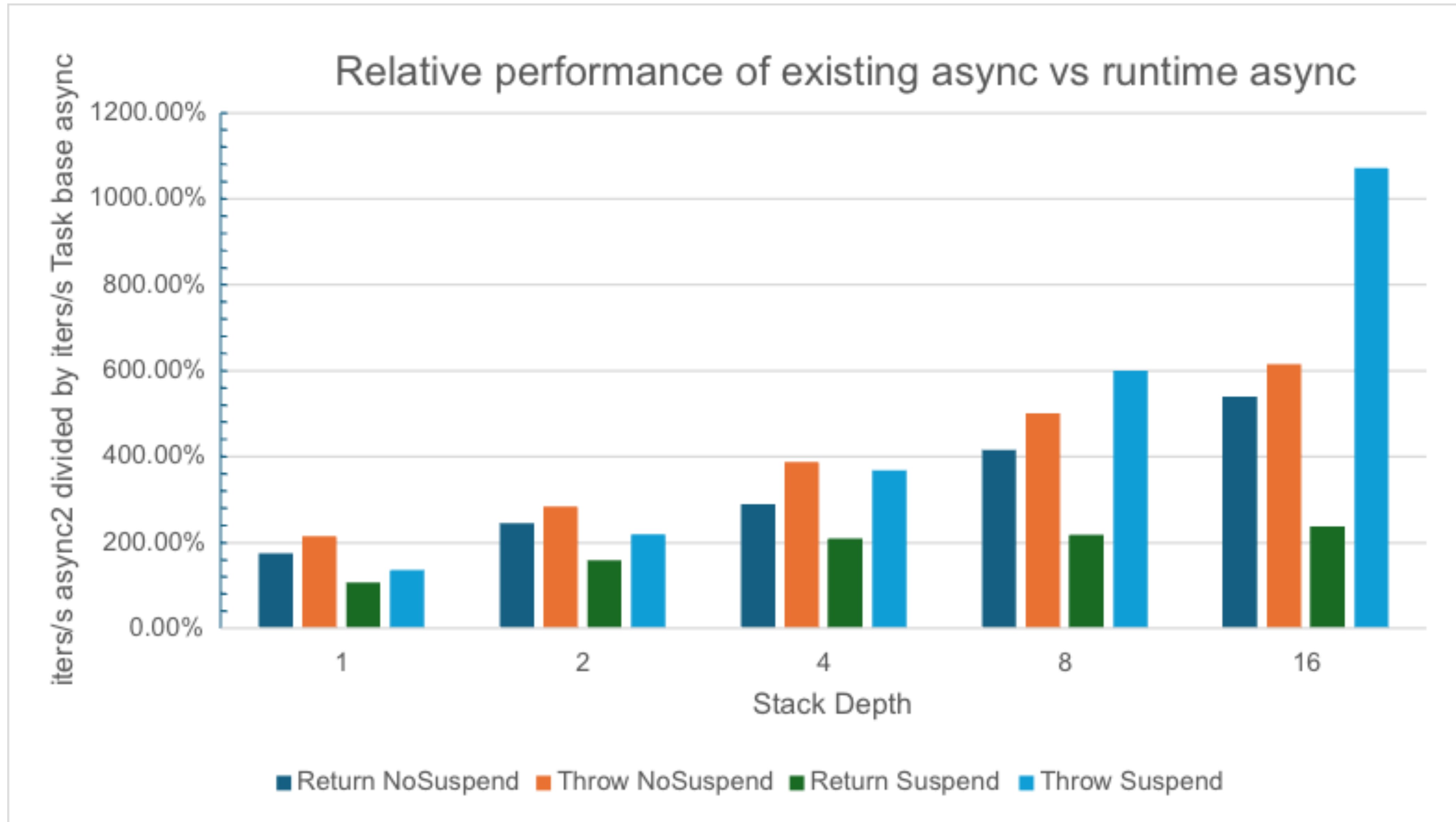
github.com/dotnet/runtimelab/blob/feature/green-threads/docs/design/features/greenthreads.md
github.com/dotnet/runtimelab/issues/2398

t.me/epeshkblog

async2: runtime-handled tasks

- Компилятор C# с поддержкой **async2** keyword
  - github.com/dotnet/roslyn/tree/demos/async2-experiment
- Рантайм .NET, с нативной поддержкой async
  - github.com/dotnet/runtimelab/tree/feature/async2-experiment
  - Stack-capturing approach
  - JIT-based state machine approach

t.me/epeshkblog



t.me/epeshkblog

roslyn AsyncRewriter

```
static async Task<int> Async1Caller()
{
    var a = await Async1Method();
    var b = await Async1Method();
    return a + b;
}
```

```
[AsyncStateMachine(typeof (<Async1Caller>d__1))]
static Task<int> Async1Caller()
{
  <Async1Caller>d__1 stateMachine;
  stateMachine.builder = AsyncTaskMethodBuilder<int>.Create();
  stateMachine.state = -1;
  stateMachine.builder.Start(ref stateMachine);
  return stateMachine.builder.Task;
}

struct <Async1Caller>d__1 : IAsyncStateMachine
{
  public int state;
  public AsyncTaskMethodBuilder<int> builder;
  private int __2;
  private TaskAwaiter<int> __1;

  void IAsyncStateMachine.MoveNext() {
    switch (state) { … }
  }
  void SetStateMachine(IAsyncStateMachine stateMachine);
}

// dotPeek: Low-level C#
```

t.me/epeshkblog

roslyn Async2Rewriter

```
static async2 Task<int> Async2Caller()
{
    var a = await Async2Method();
    var b = await Async2Method();
    return a + b;
}
```

```
static int Async2Caller()
{
  return Async2Method() + Async2Method();
}

.method public hidebysig static int32 modopt
([System.Runtime]System.Threading.Tasks.Task`1)
  Async2Caller() cil managed
```

- Async2-метод компилируется в метод, аналогичный синхронному
- Больше возможностей для JIT-оптимизаций
- Для новых оптимизаций достаточно обновить рантайм, перекомпиляция не нужна

- Однако, несмотря на IL-представление, для C# метод возвращает Task<int>

async2: code size

- async2 не разворачивает код в state machine
- IL кода получается меньше, на уровне синхронного метода

Вид методов	Размер DLL с 1000 методов
Синхронные	53 KB
async	471 KB
async2	91 KB

t.me/epeshkblog

Совместимость с async

Совместимость в обе стороны:

- `async2` может ожидать `(await) Task`
- `async` метод может ожидать `async2` метод

t.me/epeshkblog

async2 awaiting async1

```
public static int Async2Method()
{
    TaskAwaiter awaiter = Task.Delay(1).GetAwaiter();
    if (!awaiter.IsCompleted)
      // jit intrinsic
      RuntimeHelpers.UnsafeAwaitAwaiterFromRuntimeAsync<TaskAwaiter>(awaiter);

    awaiter.GetResult();
    return 2;
}
```

t.me/epeshkblog

async awaiting async2

- С точки зрения C# отличий нет – вызывается метод, возвращающий Task
- В рантайме вызывается обёртка (thunk) над async2 методом

t.me/epeshkblog

async2: fallback к task

```
static async2 Task<int> Async2Caller()
{
    var task = Async2Method();
    return await task;
}
```

- Если async2 метод не await'ится напрямую, используется fallback к вызову async2 -> async1

t.me/epeshkblog

async2: interface call

```
interface IInterface {
  public Task DoAsync();
}

public static async2 Task Caller(IInterface obj) {
    await obj.DoAsync();
}
```

- async – деталь реализации метода, а не часть интерфейса
- Caller ничего не знает про реализацию DoAsync и будет ожидать `Task` через `.GetAwaiter()`
- Двойное оборачивание! `async2(async1(DoAsync()))`

t.me/epeshkblog

async2: interface call

```
for (int i = 0; i < N; i++) {
    await obj.DoAsync();
}
```

Method	100M empty	100K Task.Delay
async2 -> async2 direct	21 ms	53 ms
async2 -> async2 interface	760 ms	120 ms
async1 -> async1 direct	560 ms	72 ms
async1 -> async1 interface	970 ms	75 ms

- `RuntimeHelpers.FinalizeValueTaskReturningThunk`
- Скорее всего, к релизу доработают и JIT сможет заменять на прямой вызов async2
- Возможно, отдельный концепт async2-делегатов и method pointers

t.me/epeshkblog

Async2 exception throwing

Method	ThrowDepth	Mean
Sync	1	5 us
Async	1	14 us
Async2 (Jit)	1	26 us
Sync	10	11 us
Async	10	65 us
Async (Jit)	10	27 us
Sync	100	75 us
Async	100	650 us
Async2 (Jit)	100	75 us
Sync	1000	705 us
Async	1000	22867 us
Async2 (Jit)	1000	650 us
Sync	10000	7791 us
Async	10000	NA
Async2 (Jit)	10000	6500 us

t.me/epeshkblog

async2 breaking changes

Synchronization/ExecutionContext не восстанавливается автоматически

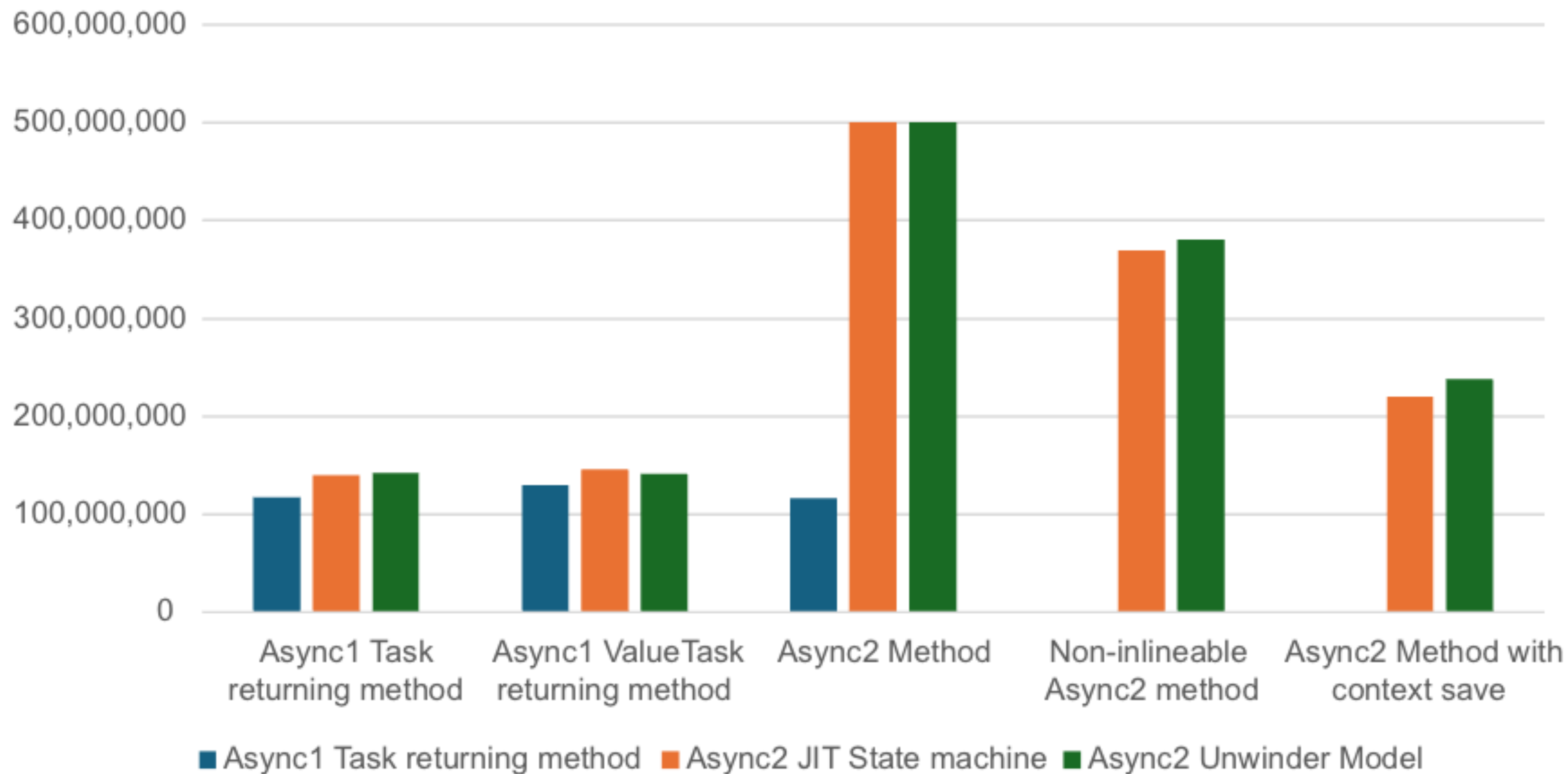
Восстанавливать контекст теперь нужно явно: `try/finally`

```
AsyncLocal<string> scope = new();

static asyncX Task A() {
    scope.Value = "A";
    await B();
    Console.WriteLine($"Scope after call: {context.Value}");
}

static asyncX Task B() {
    scope.Value = "B";
}
```

t.me/epeshkblog

AsyncLocal & IAsyncEnumerable

```
AsyncLocal<string> context = new();

async IAsyncEnumerable<int> IntSequence()
{
  context.Value = "C";
  yield return 10;
  Console.WriteLine("context.Value: {0}", context.Value);
}
```


Iterations/second calls to async methods
600,000,000
500,000,000
400,000,000
300,000,000
200,000,000
100,000,000
0
Async1 Task returning method
Async1 ValueTask returning method
Async2 Method
Non-inlineable Async2 method
Async2 Method with context save
Async1 Task returning method
Async2 JIT State machine
Async2 Unwinder Model

t.me/epeshkblog

async2 ConfigureAwaitAttribute

```
[AttributeUsage(Assembly | Module | Class | Method)]
public class ConfigureAwaitAttribute(bool continueOnCapturedContext) : Attribute { }
```

- С async2 больше не придётся вызывать .ConfigureAwait(false)
- Пока не реализовано в прототипе

async2: решаемые проблемы

Дизайн:

- Красно-синие функции
- Простор для багов
- Дублирование кода (Async-suffix)
- ConfigureAwait(false)
- CancellationToken

Технические:

- Ограниченность оптимизаций
- Аллокации
- Обработка исключений
- Размер IL-кода

t.me/epeshkblog

async2 in runtime

Две реализации

- Сохранение состояния стека (stack unwinder)
- State machine, сгенерированная на уровне JIT

```
DOTNET_RuntimeAsyncViaJitGeneratedStateMachines
```

t.me/epeshkblog

async2 stack unwinder

- До ухода в асинхронное ожидание код выполняется полностью как синхронный, без создания вспомогательных структур
- Tasklet – backup стекфрейма и регистров одного метода

```
struct Tasklet
{
    Tasklet* pTaskletNextInStack;
    Tasklet* pTaskletNextInLiveList;
    Tasklet* pTaskletPrevInLiveList;
    uint8_t* pStackData;
    uintptr_t restoreIPAddress;
    StackDataInfo* pStackDataInfo;
    TaskletReturnType taskletReturnType;
    // min generation of all managed objects referred from this frame.
    // -1 means the frame is a part of actiely executing stack and may have
byrefs pointing to it
    int32_t  minGeneration;
    Tasklet* pTaskletPrevInStack;
};
```

t.me/epeshkblog

async2 stack unwinder

- До ухода в асинхронное ожидание код выполняется полностью как синхронный, без создания вспомогательных структур
- Tasklet – backup стекфрейма и регистров одного метода
- При переходе в асинхронное ожидание (suspend) методы снимаются со стека и сохраняются в tasklet
- Возобновление (resumption) методов происходит по одному

t.me/epeshkblog

async2 stack unwinder

```
async2 Task A() {
  await B();
}

async2 Task B() {
  Console.Write(1); ←
  await Task.Yield();
  Console.Write(2);
  await Task.Yield();
}
```

- A()
  - B()
    - Write(1)

t.me/epeshkblog

async2 stack unwinder

```
async2 Task A() {
  await B();
}

async2 Task B() {
  Console.Write(1);
  await Task.Yield(); ←
  Console.Write(2);
  await Task.Yield();
}
```

- A()
  - B()
    - Yield()
      - await
        - RuntimeSuspension_CaptureTasklets

t.me/epeshkblog

async2 stack unwinder

```
async2 Task A() {
  await B();
}

async2 Task B() {
  Console.Write(1);
  await Task.Yield(); ←
  Console.Write(2);
  await Task.Yield();
}
```

- empty

```
A() {
  B();
  ←
}
```

```
B() {
  Write("B1");
  Yield();
  ←
  Write("B2");
  Yield();
  Write("B3");
}
```

t.me/epeshkblog

async2 stack unwinder

```
async2 Task A() {
  await B();
}

async2 Task B() {
  Console.Write(1);
  await Task.Yield();
  Console.Write(2); ←
  await Task.Yield();
}
```

- B()
  - Write(2)

```
A() {
  B();
  ←
}
```

```
B() {
  Write("B1");
  Yield();
  ←
  Write("B2");
  Yield();
  Write("B3");
}
```

A() на стеке пока нет!

t.me/epeshkblog

async2 stack unwinder

```
async2 Task A() {
  await B();
}

async2 Task B() {
  Console.Write(1);
  await Task.Yield();
  Console.Write(2);
  await Task.Yield(); ←
}
```

- B()
  - Yield
    - await

```
A() {
  B();
  ←
}
```

```
B() {
  Write(1);
  Yield();
  Write(2);
  Yield();
  ←
}
```

t.me/epeshkblog

async2 stack unwinder

```
async2 Task A() {
  await B();
}

async2 Task B() {
  Console.Write(1);
  await Task.Yield();
  Console.Write(2);
  await Task.Yield();
} ←
```

- B()

```
A() {
  B();
  ←
}
```

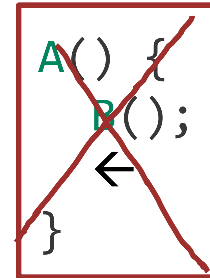
```
B() {
  Write(1);
  Yield();
  Write(2);
  Yield();
  ←
}
```

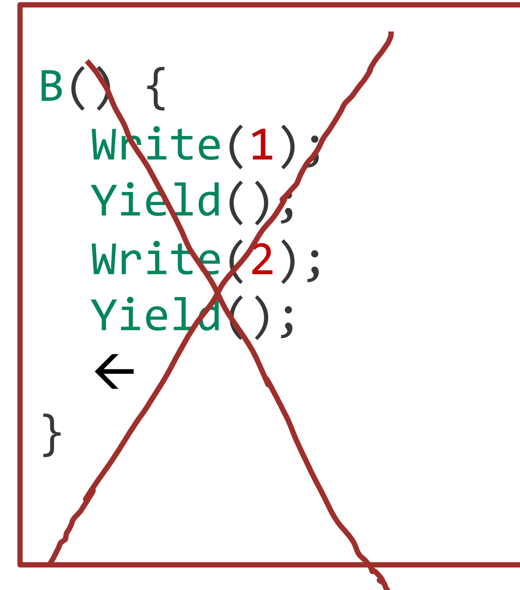
t.me/epeshkblog

async2 stack unwinder

```
async2 Task A() {
  await B();
} ←

async2 Task B() {
  Console.Write(1);
  await Task.Yield();
  Console.Write(2);
  await Task.Yield();
}
```

- A()





t.me/epeshkblog

async2 stack unwinder: недостатки

- GC паузы – в 4 раза больше async1
- Потребление памяти на стекфреймы – в 3 раза больше async1
- Требуются доработки не только JIT, но и GC

t.me/epeshkblog

async2 JIT state machine

- Аналог нынешней реализации, но state machine генерируется JIT
- В отличие от tasklets, managed подход, не требует кооперации с GC

```
internal sealed unsafe class Continuation
{
    public Continuation? Next;
    public delegate*<Continuation, Continuation?> Resume;
    public uint State;
    public CorInfoContinuationFlags Flags;
    public byte[]? Data; // <- локальные переменные
    public object[]? GCData; // <- локальные переменные, reference types
}
```

t.me/epeshkblog

Особенности прототипа

```
public static async2 Task RefAcceptingAsync2(ref int x)
{
    await Task.Delay(1);
    Console.WriteLine(x);
}
```

- Наконец-то можно использовать ref/out-параметры для async
- Такой код скомпилируется в прототипе (поддержка ref – часть эксперимента)
- Stack-capture версия: запланировано, но пока некорректное значение
- Jit state machine: `InvalidProgramException`

t.me/epeshkblog

Выводы

t.me/epeshkblog

Links

t.me/epeshkblog

Вопросы

Евгений Пешков

@epeshk

DOTNEXT