ADC

How to async await:

Internals and expert knowledge for scalable .NET Apps, by Kenny Pflug
Workshop at the Advanced Developers Conference 2025, Regensburg, Germany

TFSH TELIS FinancialServicesHolding Aktiengesellschaft

ppedv

gwvs

On today's menu

- Part 1: **The Basics**
  async await, TPL, the .NET thread pool, async I/O and async compute
- Part 2: **The async state machine** and its memory behavior during runtime
- Part 3: **IAsyncEnumerable** and **Cancellation Token Sources**
- Part 4: **Async triggering** of long-running asynchronous operations

https://github.com/feO2x/adc2025-async-await-workshop

Kenny Pflug

@feO2x

- Senior Software Developer at GWVS, TELIS FinancialServicesHolding AG
- Professional Developer for .NET since 2009
- Distributed Systems with ASP.NET (Core) since 2014
- CLR and Framework Internals (Memory Management, Asynchronous Programming, Threading, ORMs, DI Containers, Serializers, etc.)

TFSH TELIS FinancialServicesHolding Aktiengesellschaft

gwvs

Demo Time!

async await and service scalability

Part 1: The Basics

async await, Task-Parallel Library (TPL), the .NET Thread Pool, async I/O and async compute

ppedv

Demo Time!

From sync to async

The basics of async await

- If you want to use the **await** keyword, you need to add the **async** modifier to the containing method (this works for anonymous methods, too).
- Each **async** method is converted to a state machine.
- **await** needs to be called on a **task** (or a task-like object) which represents an asynchronous operation.
- When the **await** keyword is executed, the method returns to its caller, unless the task completes synchronously.
- When the task ran asynchronously and signals completion, the **continuation** is enqueued either on the .NET Thread Pool or on the original calling thread if it has a **SynchronizationContext**.
- The **continuation** are all expressions/statements that are executed after the **await** keyword – it can contain other **await** expressions.
- While **void** is a valid return type for async methods, callers will not be able to await the state machine. By default, you should return a **task** type.

Why should we use async await?

- In my opinion, it is the **easiest, most readable and most maintainable way** to write asynchronous code.
- Remember the hassle with the old Asynchronous Programming Model (APM) with its **BeginXXX** and **EndXXX** methods as well as **IAsyncResult**? Each continuation needed to be its own method.
- async await works asynchronously but **reads like synchronous code**.
- **Debuggers** let you **step over await expressions** as if they were synchronous.

Two kinds of async operations

Async I/O

1. On the calling thread, we start an I/O operation.
2. At some point, a controller takes over (usually network or disk) and we return to the caller.
3. Once the I/O operation is done, the continuation is enqueued.

Async Compute

1. We do not want to execute a computation on the current thread, so, we move it to another thread.
2. The async computation is represented by a task, we return to the caller.
3. Once the computation is done, the continuation is enqueued.

Tasks in the Task Parallel Library (TPL)

- **Task** and **Task<T>** are the main objects that you will await in your code.
- They can represent both async I/O (usually via **TaskCompletionSource**, encapsulated by Data Access Libraries) as well as async compute (usually via **Task.Start/Task.Run**).
- They will signal completion via various APIs (for async await, the most important one is **TaskAwaiter**).
- **ValueTask** and **ValueTask<T>** can be used when you need to optimize performance but be aware that they can only be awaited once.
- You can also implement your own tasks, but we won't address this here (see talk from Peter Butzhammer on ADC Tuesday).

Hands on!

Avoid Blocking I/O Calls

Hands on:
Introduce async await to two endpoints

- The **DeleteContactEndpoint** is currently implemented synchronously.
- Identify the I/O calls and replace them with their asynchronous counterparts.
- The endpoint accesses the database via Entity Framework Core.

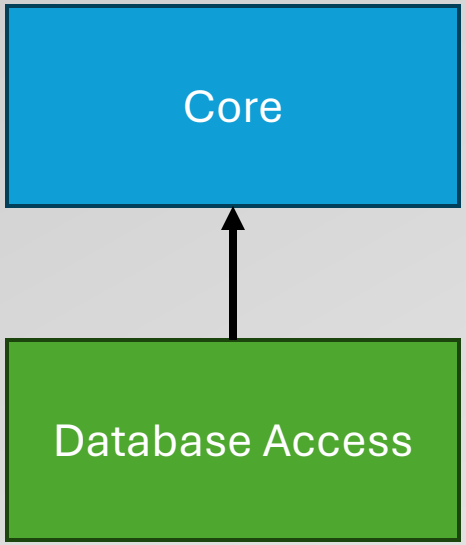
cd 03-Playground

docker compose up -d

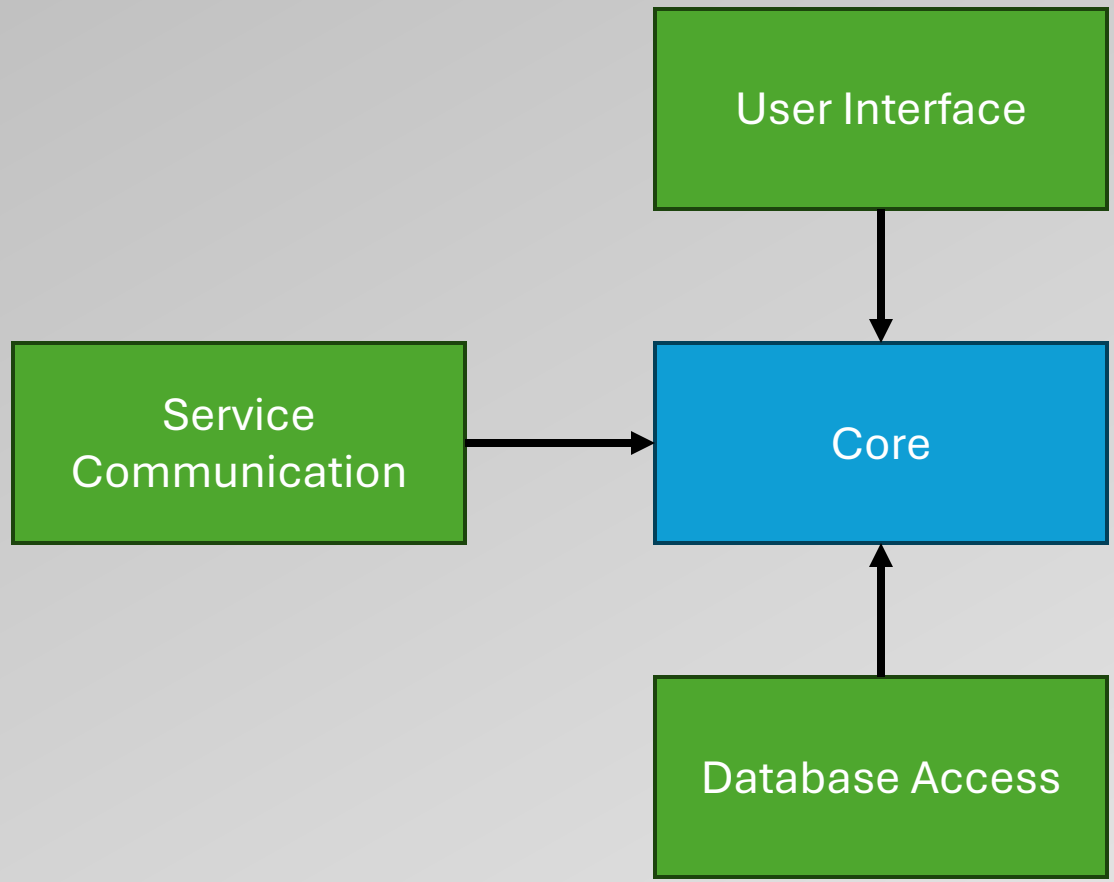
Asynchronous members in interfaces (1)

Core

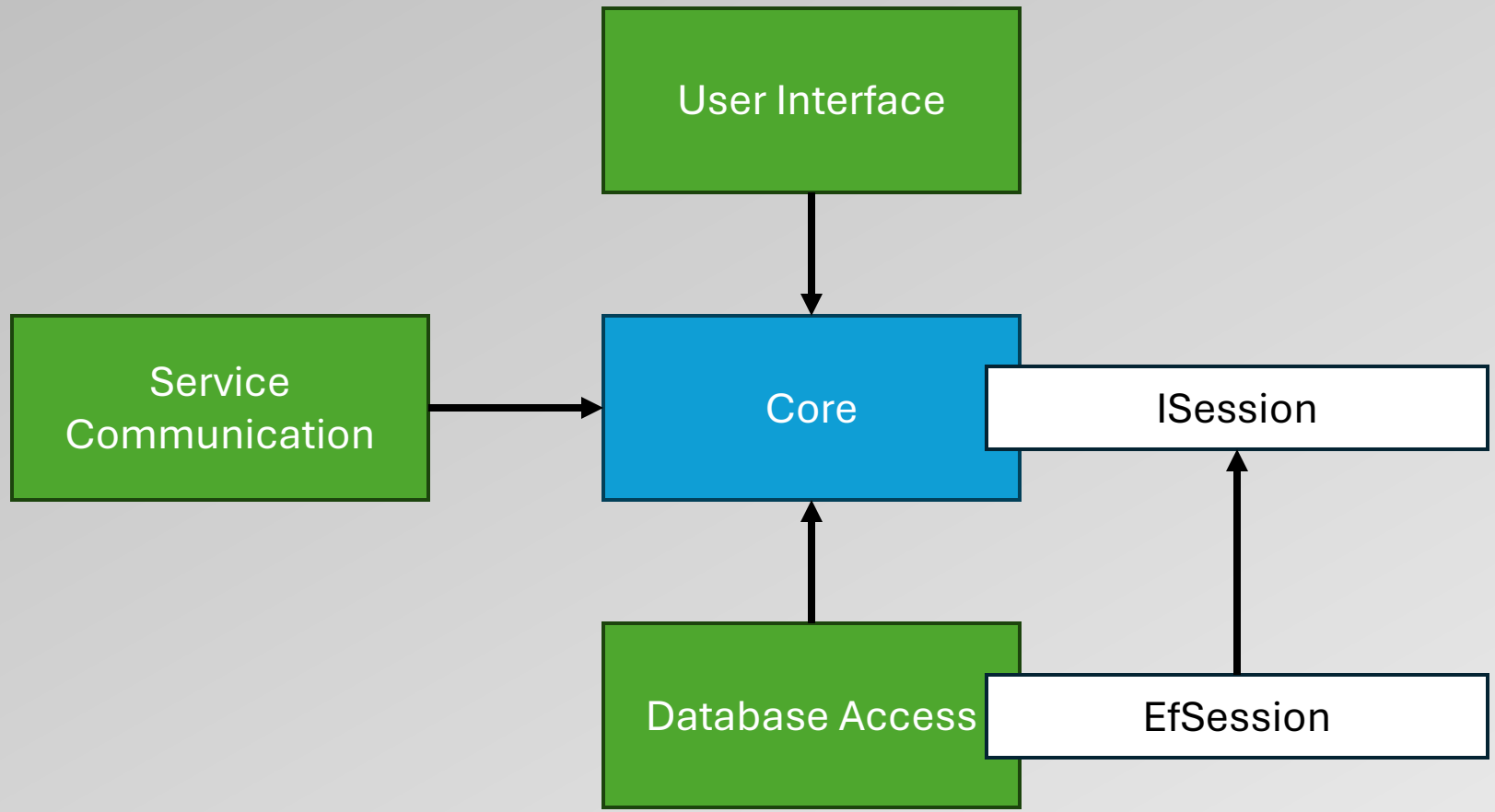
Asynchronous members in interfaces (2)



Asynchronous members in interfaces (3)



Asynchronous members in interfaces (4)



Asynchronous members in interfaces (5)

```
public interface IGetContactSession
{
    Contact? LoadById(Guid id);
}

public class EfGetContactSession : IGetContactSession
{
    public Contact? LoadById(Guid id) { }
}
```

Asynchronous members in interfaces (6)

```
public interface IGetContactSession
{
    Contact? LoadById(Guid id);
}

public class EfGetContactSession : IGetContactSession
{
    public async Task<Contact?> LoadByIdAsync(
        Guid id,
        CancellationToken cancellationToken = default
    ) { }
}
```

When we want to make a method asynchronous...

Asynchronous members in interfaces (7)

```
public interface IGetContactSession
{
    Task<Contact?> LoadByIdAsync(
        Guid id,
        CancellationToken cancellationToken = default
    );
}

public class EfGetContactSession : IGetContactSession
{
    public async Task<Contact?> LoadByIdAsync(
        Guid id,
        CancellationToken cancellationToken = default
    ) { }
}
```

...we might need to adapt the corresponding interface.

About async members in types

- The **async** keyword is never applied to interface members or abstract methods.
- Guideline: the method name should have the suffix **'Async'** when the return value is a task (or task-like type). It does not matter if an implementor uses async or not (**methods that return tasks can be implemented in a synchronous manner**).
- By default, pass a **CancellationToken** as the last parameter. Set it to **default** so callers do not have to specify a value. This makes the operation cancellable.
- Method bodies of **constructors**, **properties**, and **events** cannot contain **await** calls. They are always executed synchronously – you should not perform any I/O calls in their method bodies (especially prevalent in constructors – use a factory with an **CreateAsync** method instead).
- When you call an async method, your own method should be async, too, unless you are not interested in the results (side effects, return value) of the operation.

Hands on!

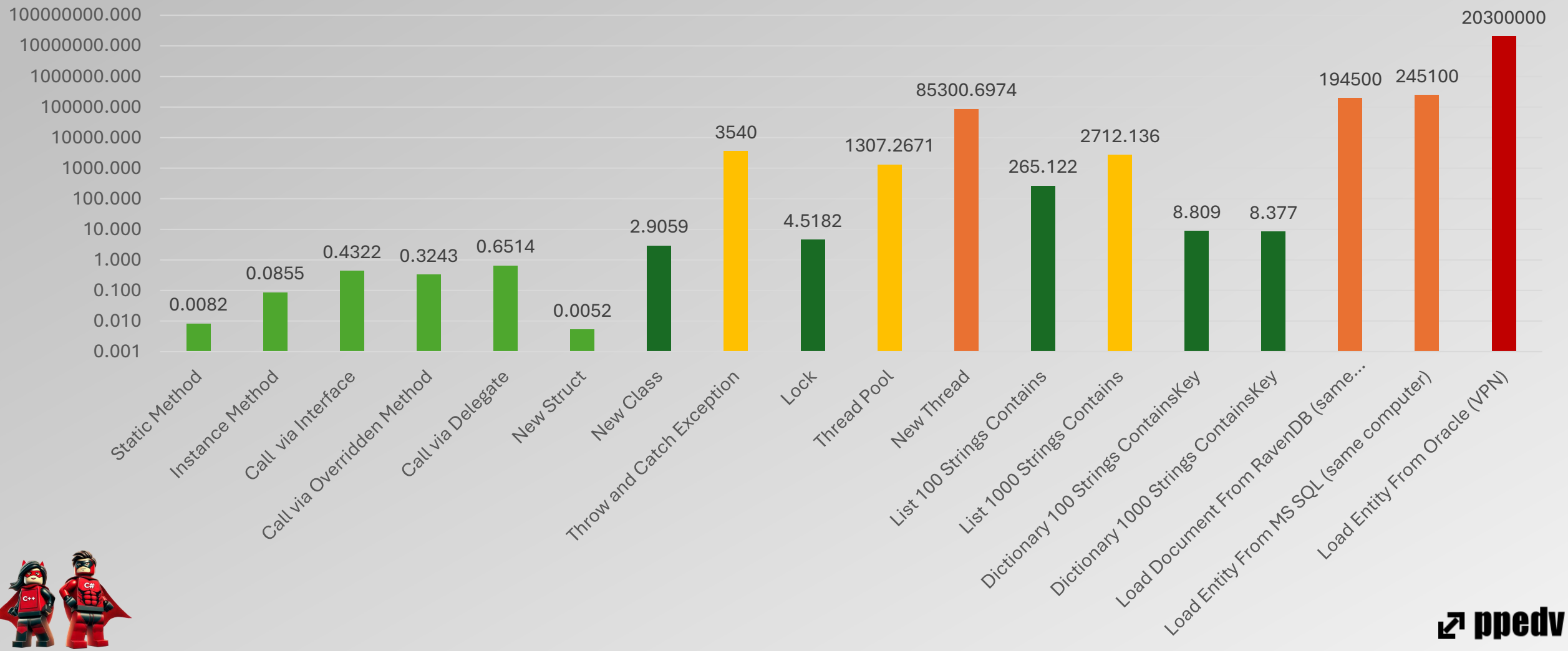
async call via abstraction

Hands on:
Create interfaces that support async await

- For the **DeleteContactEndpoint**, introduce an interface and implementation for a Humble Object to decouple Database Access from the endpoint logic.
- Light.DatabaseAccess.EntityFrameworkCore is already integrated into the project, you can use the **ISession** interface **EfSession<T>** base class for your humble object.
- Do not abstract away the **SaveChangesAsync** call from the endpoint.
- Take inspiration from the other two endpoints of the Contacts vertical slice.

Performance of everyday things

BenchmarkDotNet=v0.12.1, OS=Windows 10.0.19042
AMD Ryzen 9 5950X, 1 CPU, 32 logical and 16 physical cores
.NET Core SDK=5.0.103
[Host] : .NET Core 5.0.3 (CoreCLR 5.0.321.7212, CoreFX 5.0.321.7212), X64 RyuJIT
Job-SWHTBI : .NET Core 5.0.3 (CoreCLR 5.0.



ppedv

Threads and performance in .NET

- A thread is one of the objects with the largest impact on memory and performance.
- Thread stack default size: 1MB in 32 Bit mode, 4MB in 64 Bit mode.
- .NET Threads are associated with OS threads, which have their own kernel objects to save registers during Context Switches.
- Context Switches:
  - every ~15ms, the OS interrupts running threads and chooses new ones to run on the CPU cores.
  - a thread starting to run likely needs other data than the previous thread (CPU Cache Misses).
- On Windows: every loaded DLL is notified when a thread is instantiated and destroyed.

Avoid instantiating threads, reuse those of the Thread Pool

How many threads

- Depends on the number of CPU cores in your target system
- Ideally, you have one thread per “virtual” core: AMD Simultaneous Multi-Threading (SMT), Intel Hyper Threading
- The .NET Thread Pool manages an “optimal” number of worker threads
- It increases (and decreases) the number of threads dynamically based on the number of tasks you throw at it
- Work can be handed over e.g. via **ThreadPool.QueueUserWorkItem** or **Task.Start**

The big issue with the .NET Thread Pool

Do not block the Thread Pool's Worker Threads!
It will create more if work items need to be handled by it.

The big issue with the .NET Thread Pool

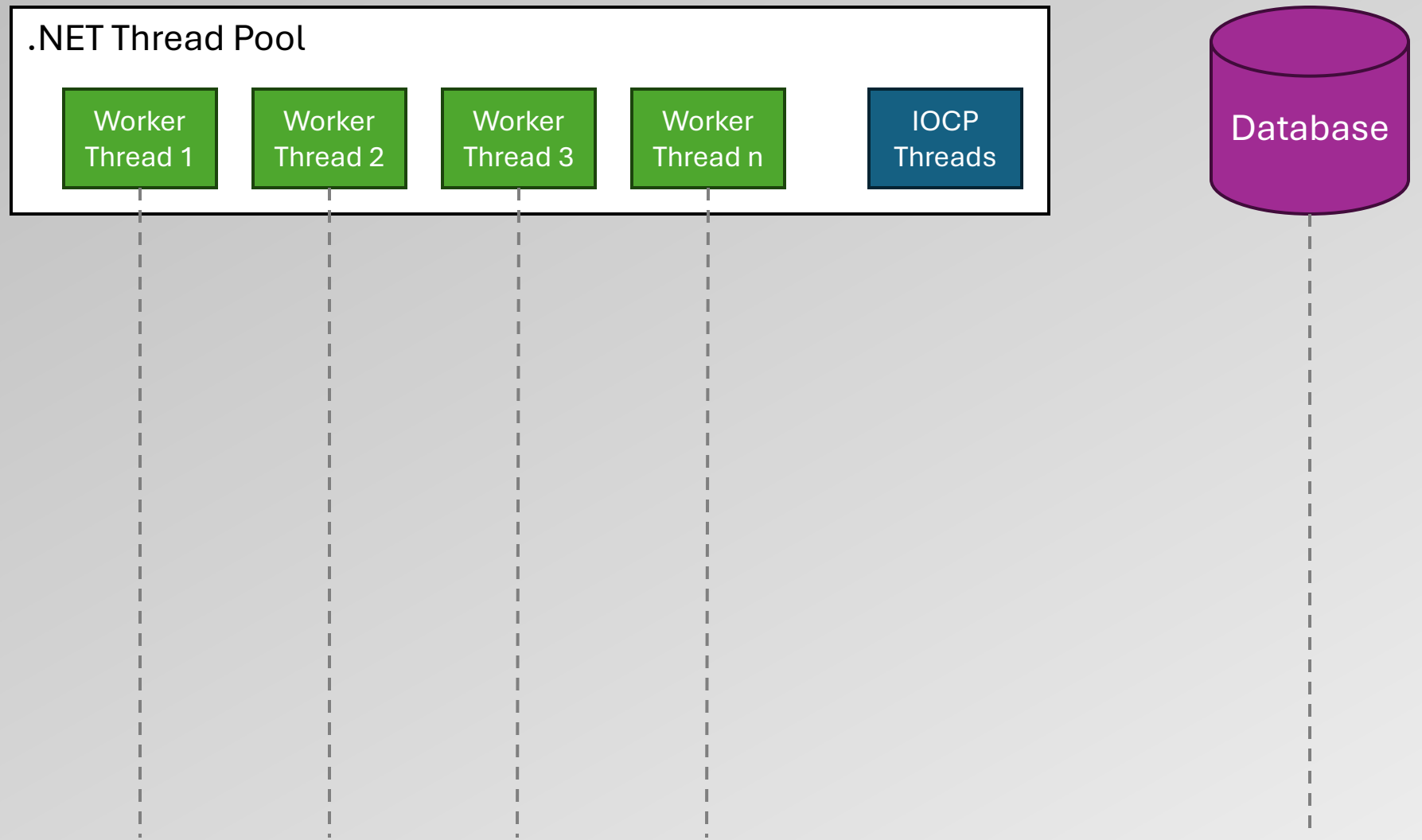
How do we block worker threads?

1. **Thread.Sleep**
2. **synchronous I/O**
3. **synchronous waiting for async compute**

The big issue in more detail

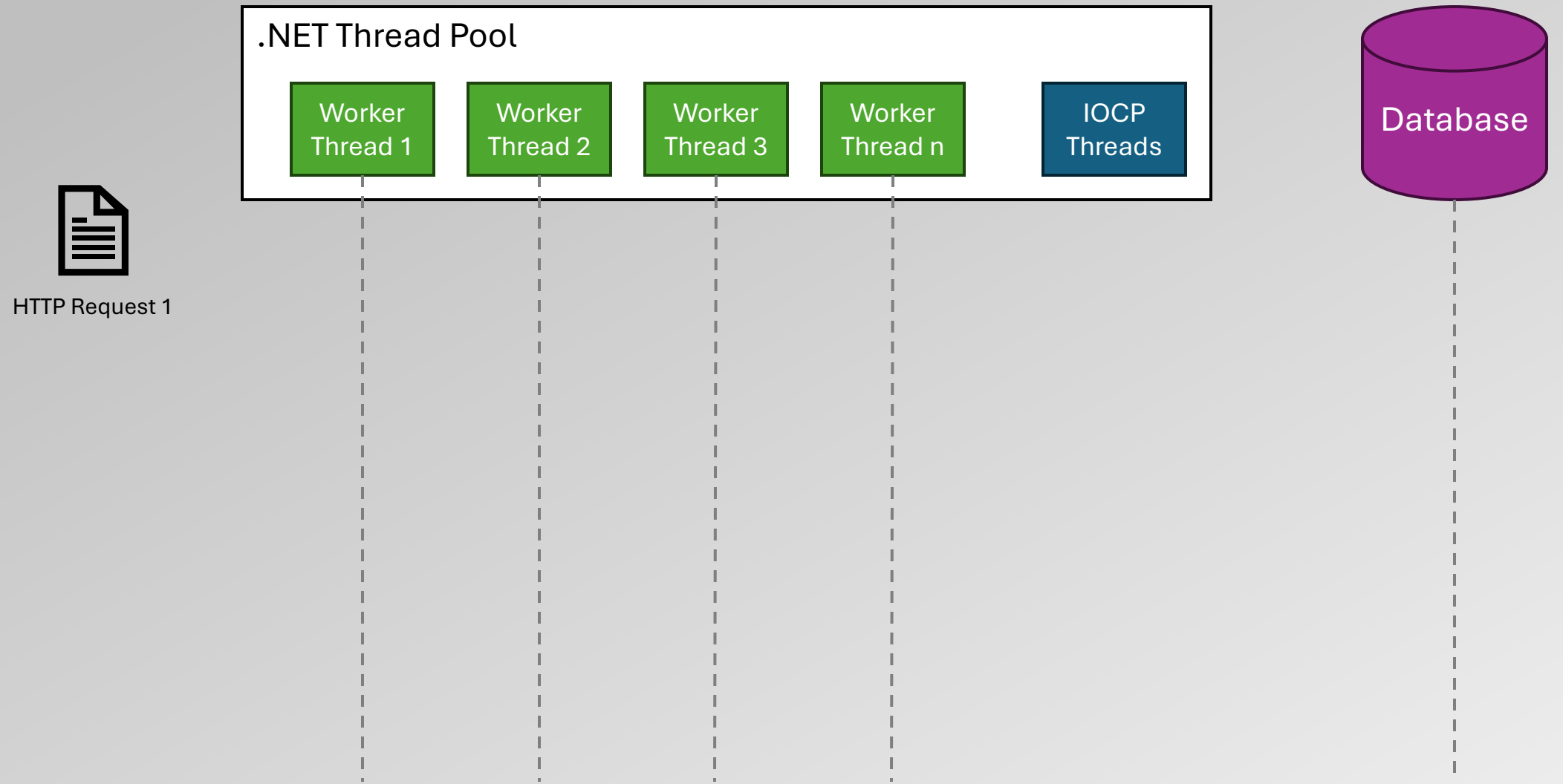
- If the .NET Thread Pool has work items in its queue and the OS signals that the pool's worker threads are blocked, the pool will create additional worker threads.
- Depending on the amount of work items and the block duration, this can spiral out of control (Thread Pool Starvation).
- Since .NET 7, the .NET Thread Pool is no longer implemented in C++, but in C# (ThreadPool.Portable in dotnet/runtime repo).
- This comes with a new Hill-Climbing algorithm for worker thread allocation which creates less worker threads and decreases its number earlier towards the goal of the CPU's logical core count.

ASP.NET Core services are inherently multi-threaded



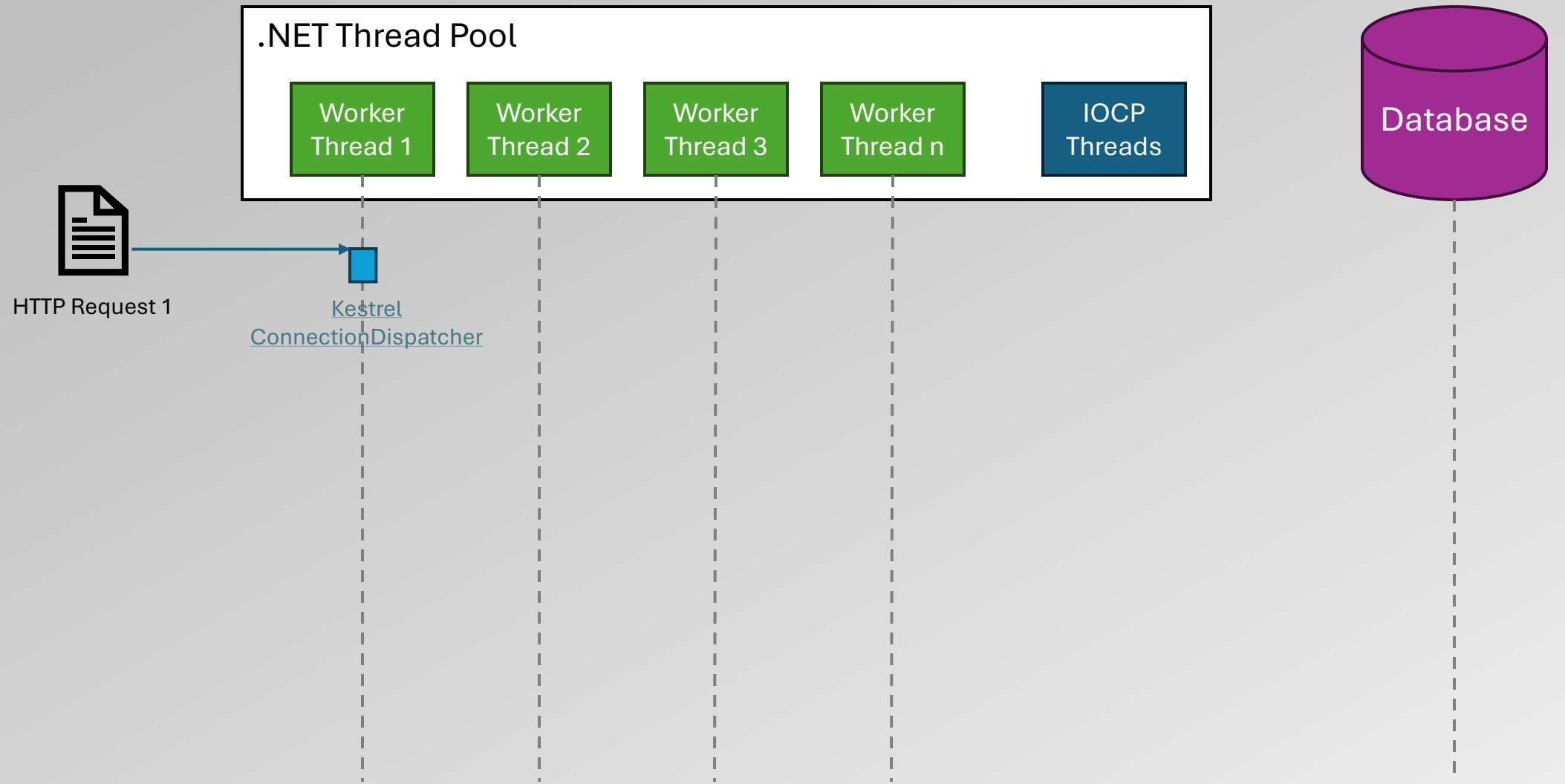
ppedv

ASP.NET Core services are inherently multi-threaded



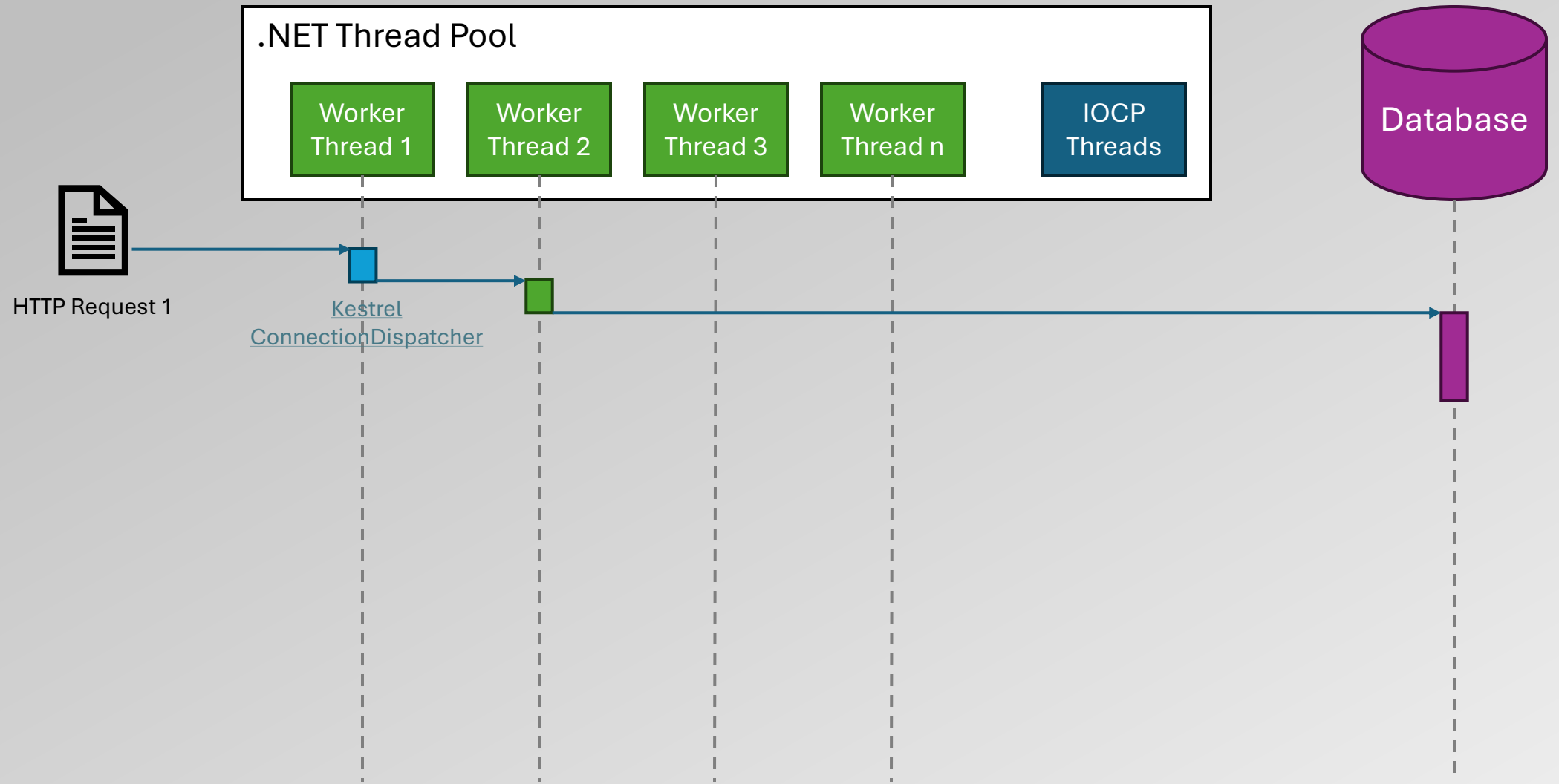
ppedv

ASP.NET Core services are inherently multi-threaded



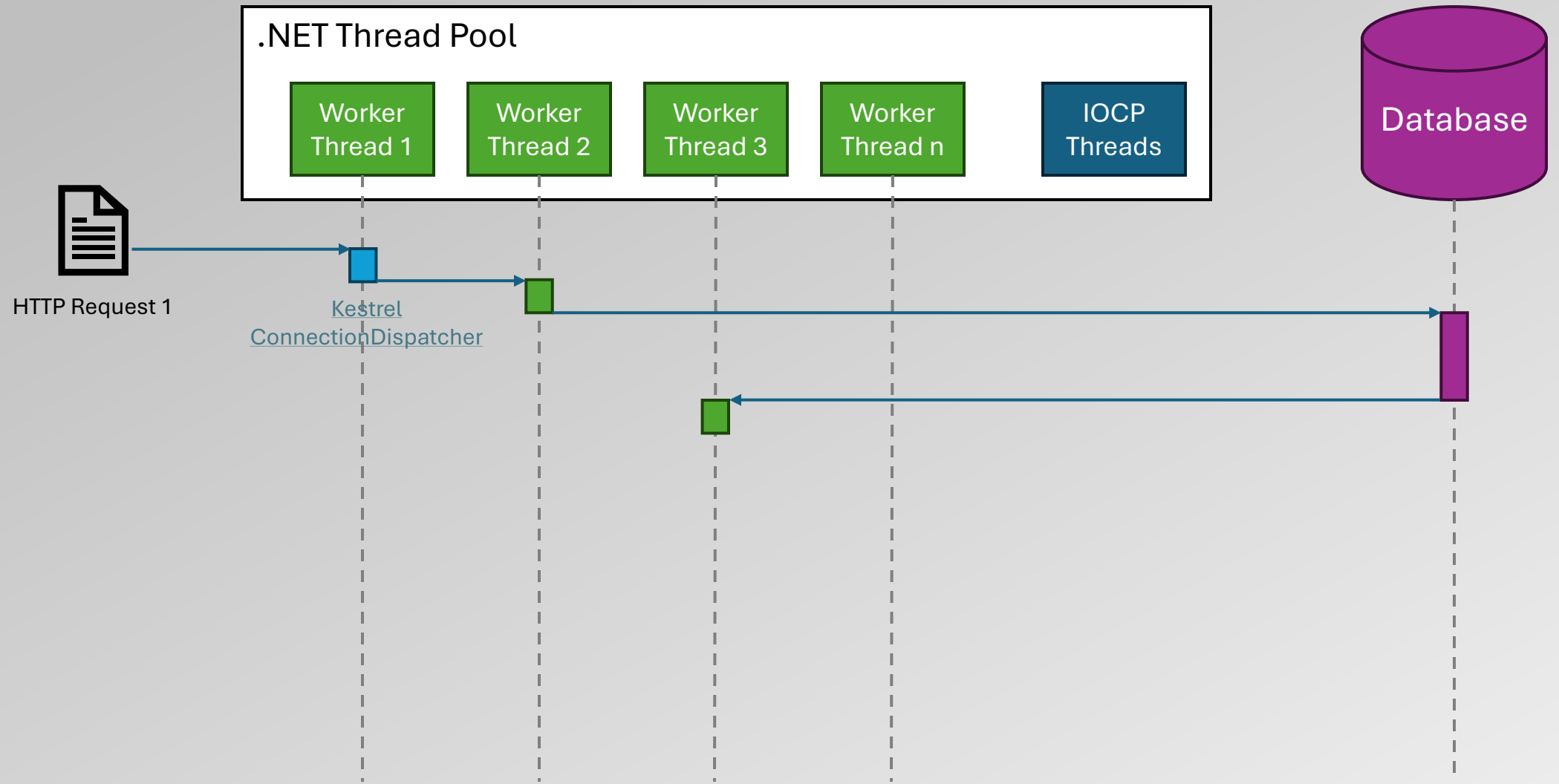
ppedv

ASP.NET Core services are inherently multi-threaded



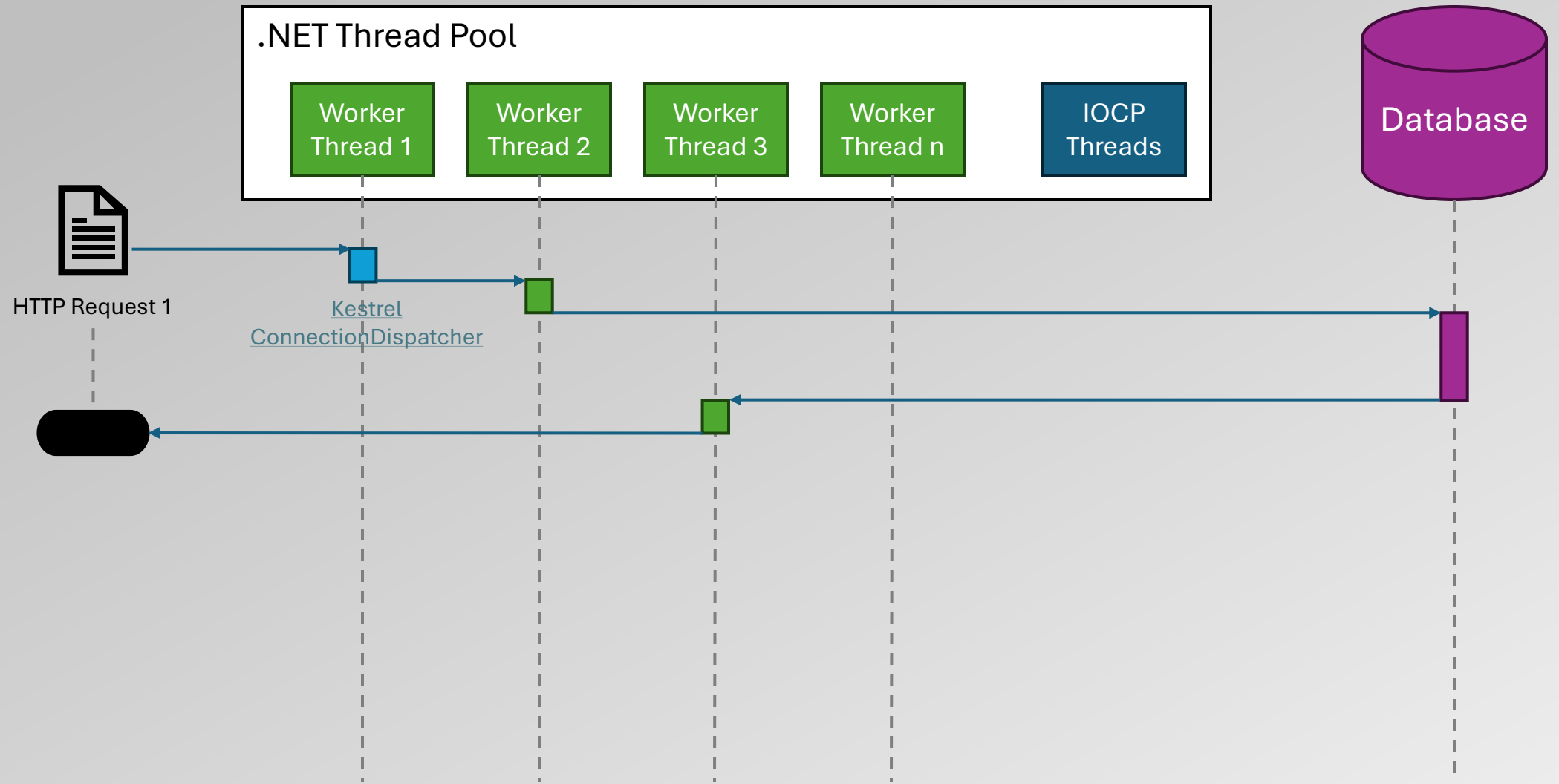
ppedv

ASP.NET Core services are inherently multi-threaded



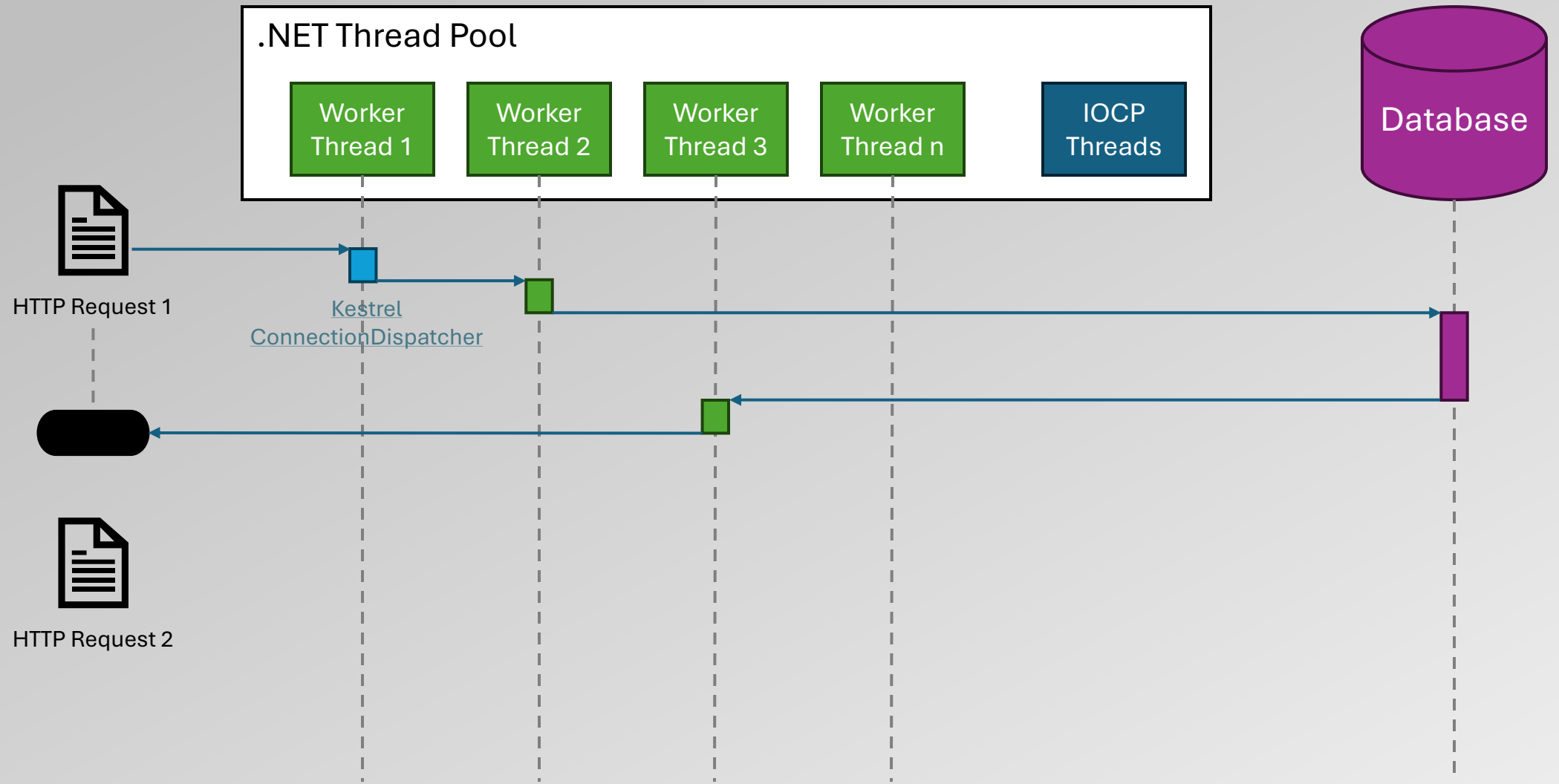
ppedv

ASP.NET Core services are inherently multi-threaded



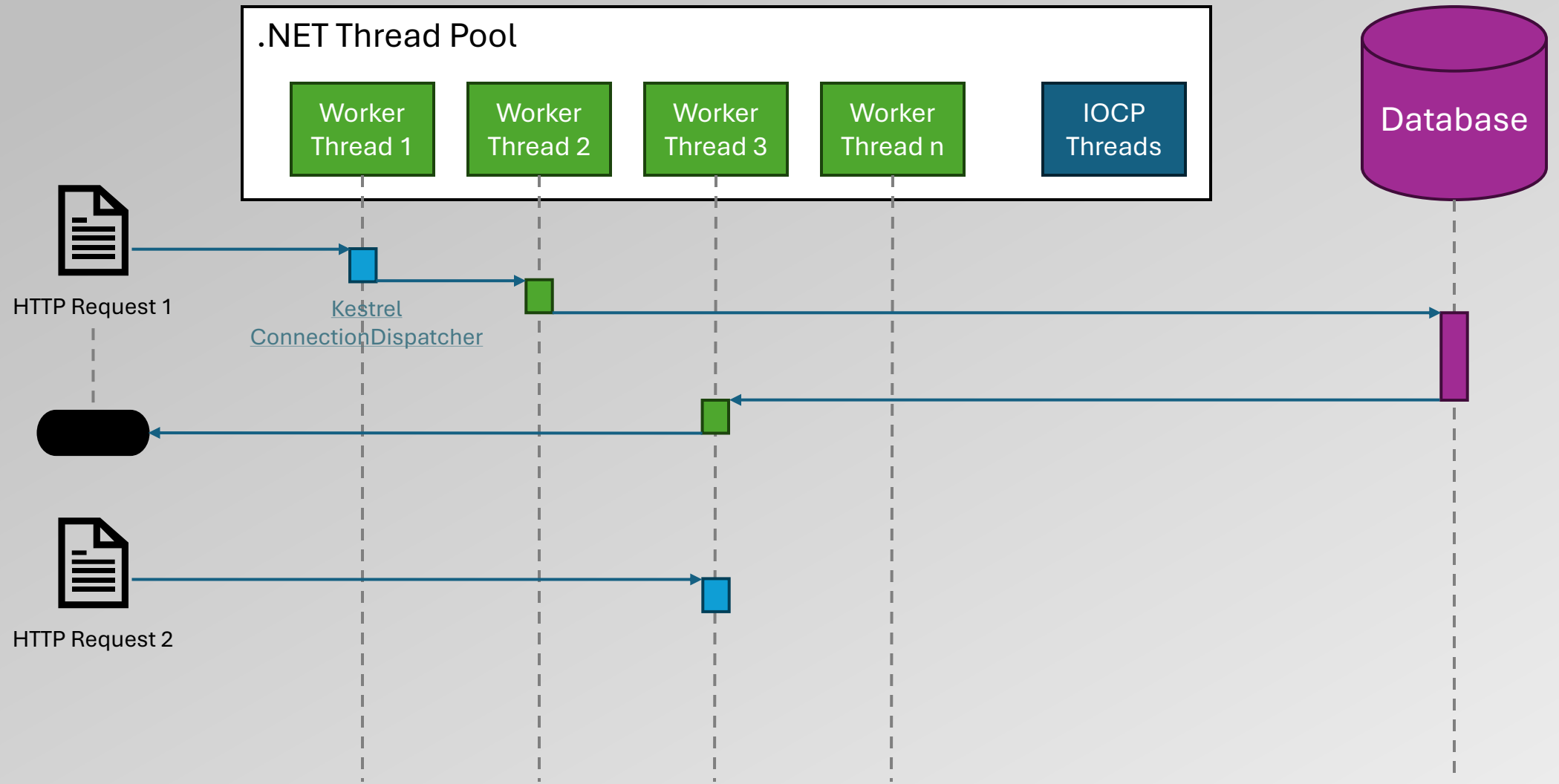
ppedv

ASP.NET Core services are inherently multi-threaded



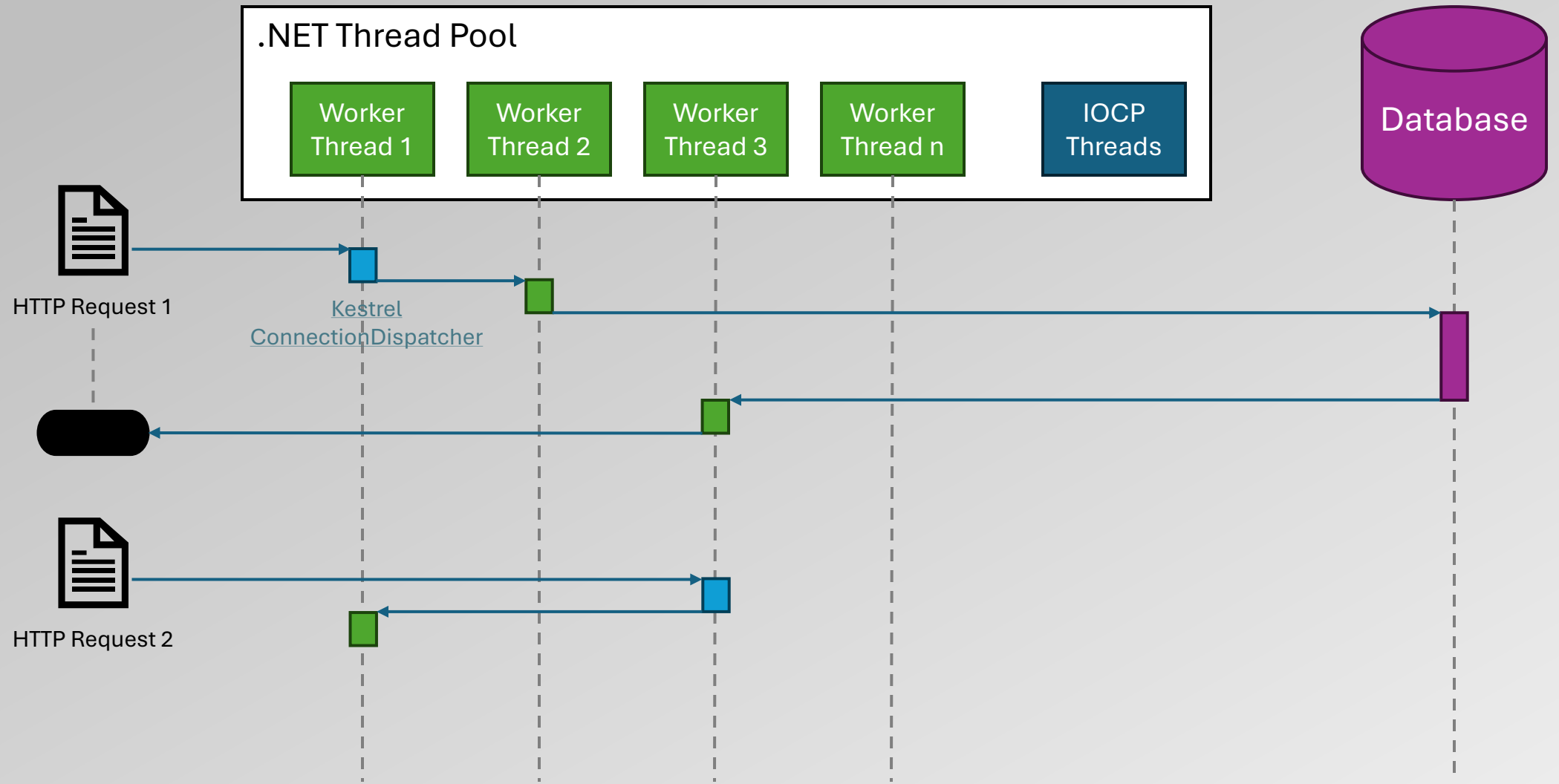
ppedv

ASP.NET Core services are inherently multi-threaded



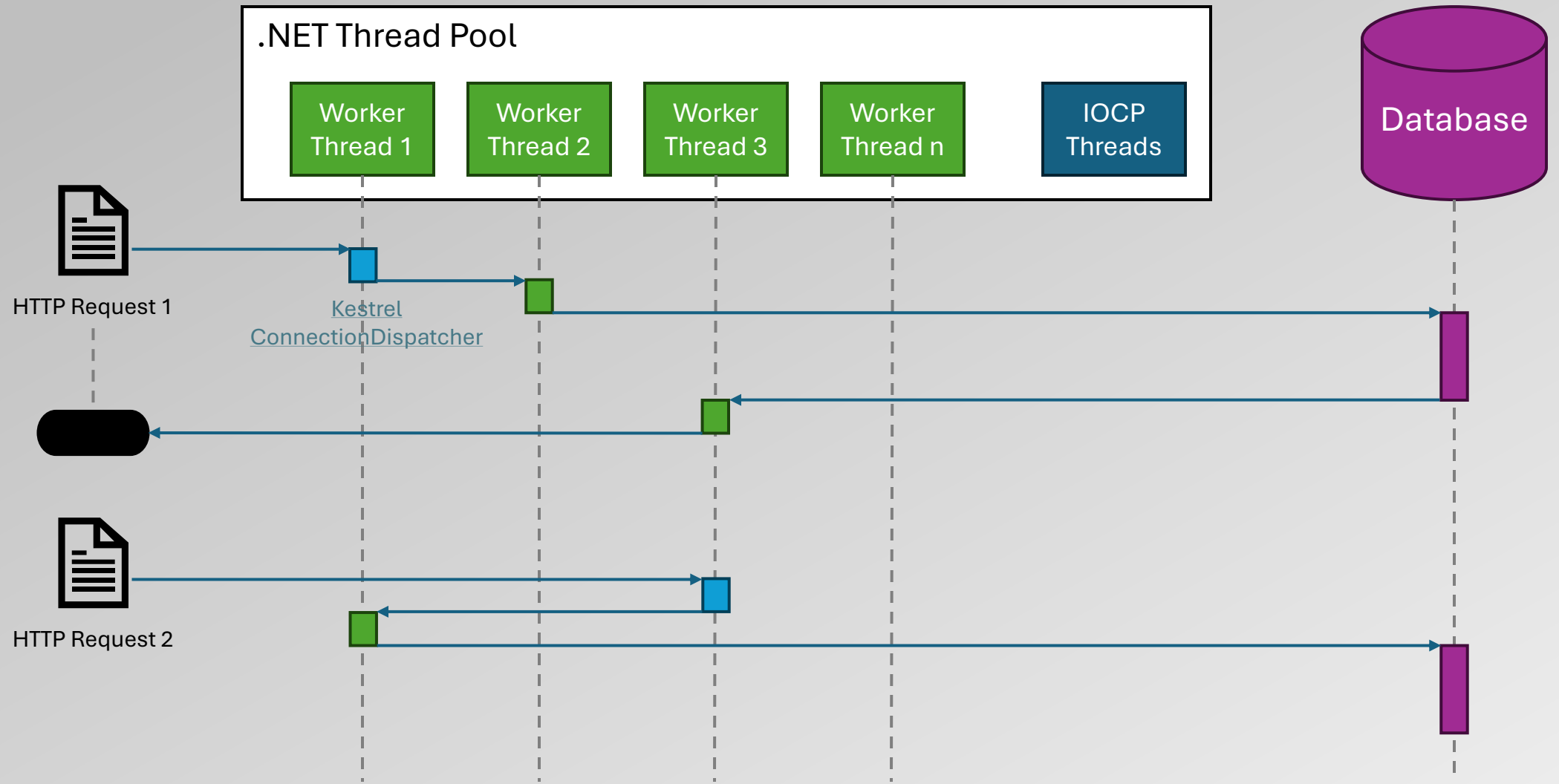
ppedv

ASP.NET Core services are inherently multi-threaded



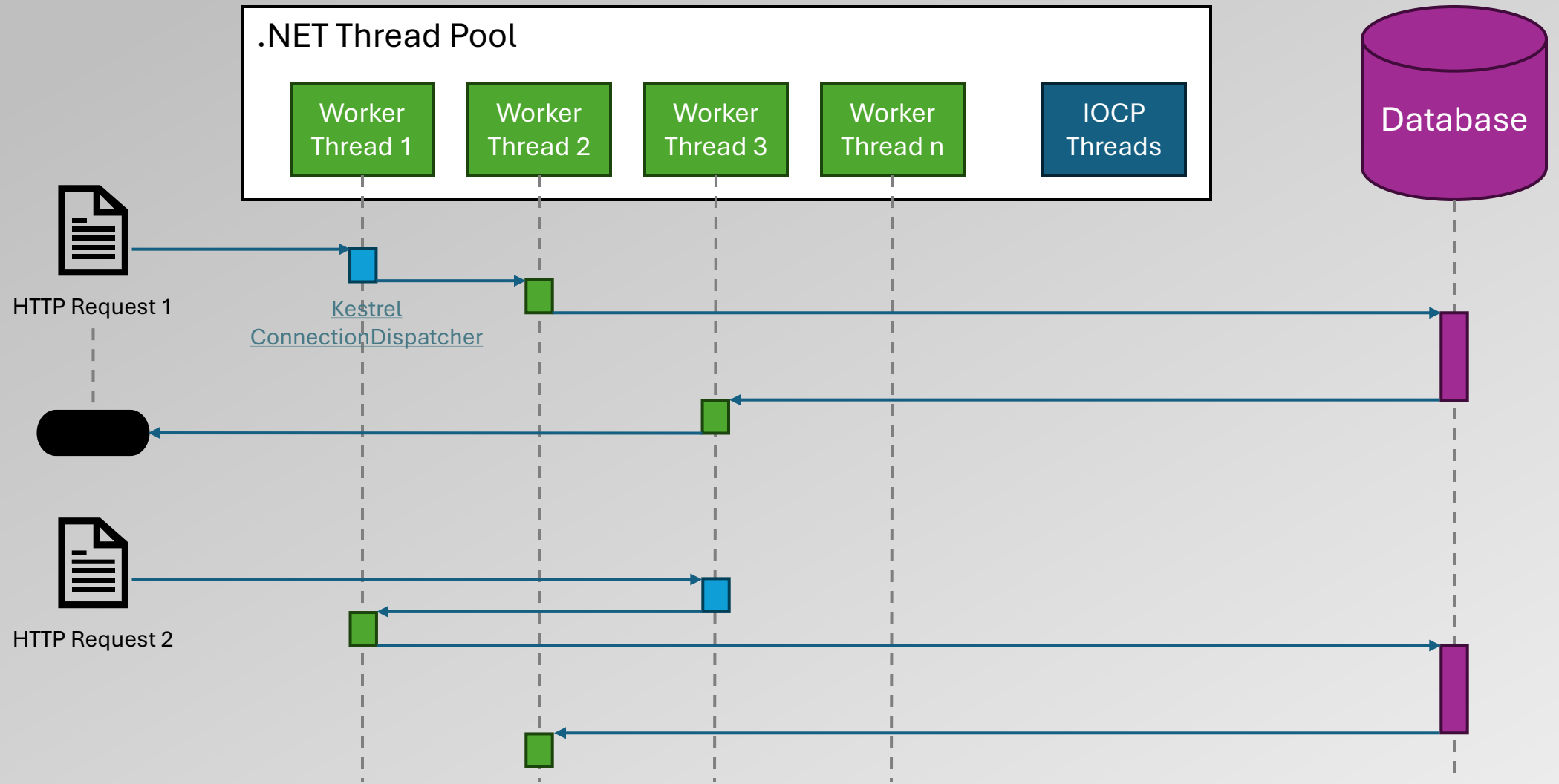
ppedv

ASP.NET Core services are inherently multi-threaded



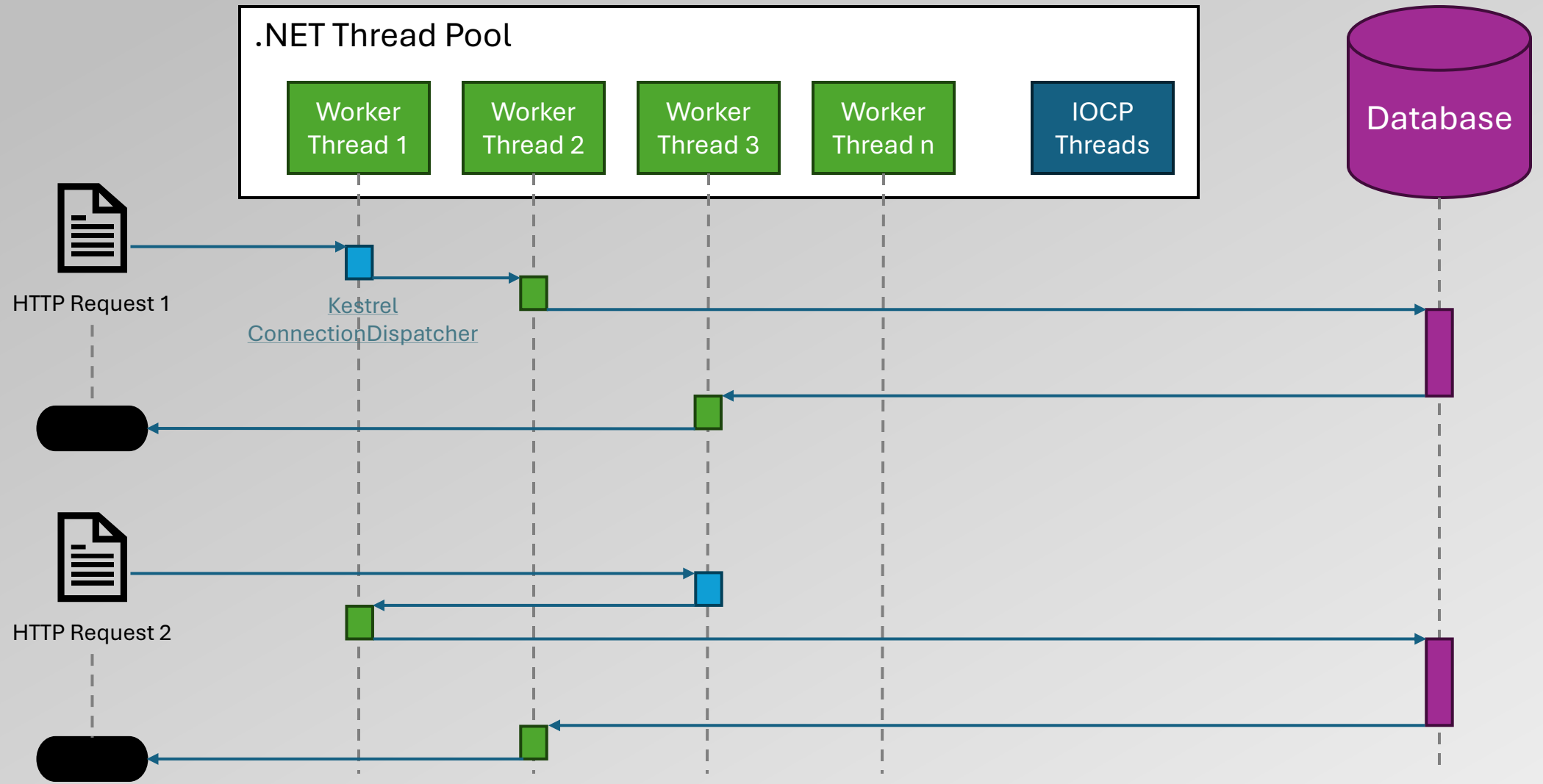
ppedv

ASP.NET Core services are inherently multi-threaded



ppedv

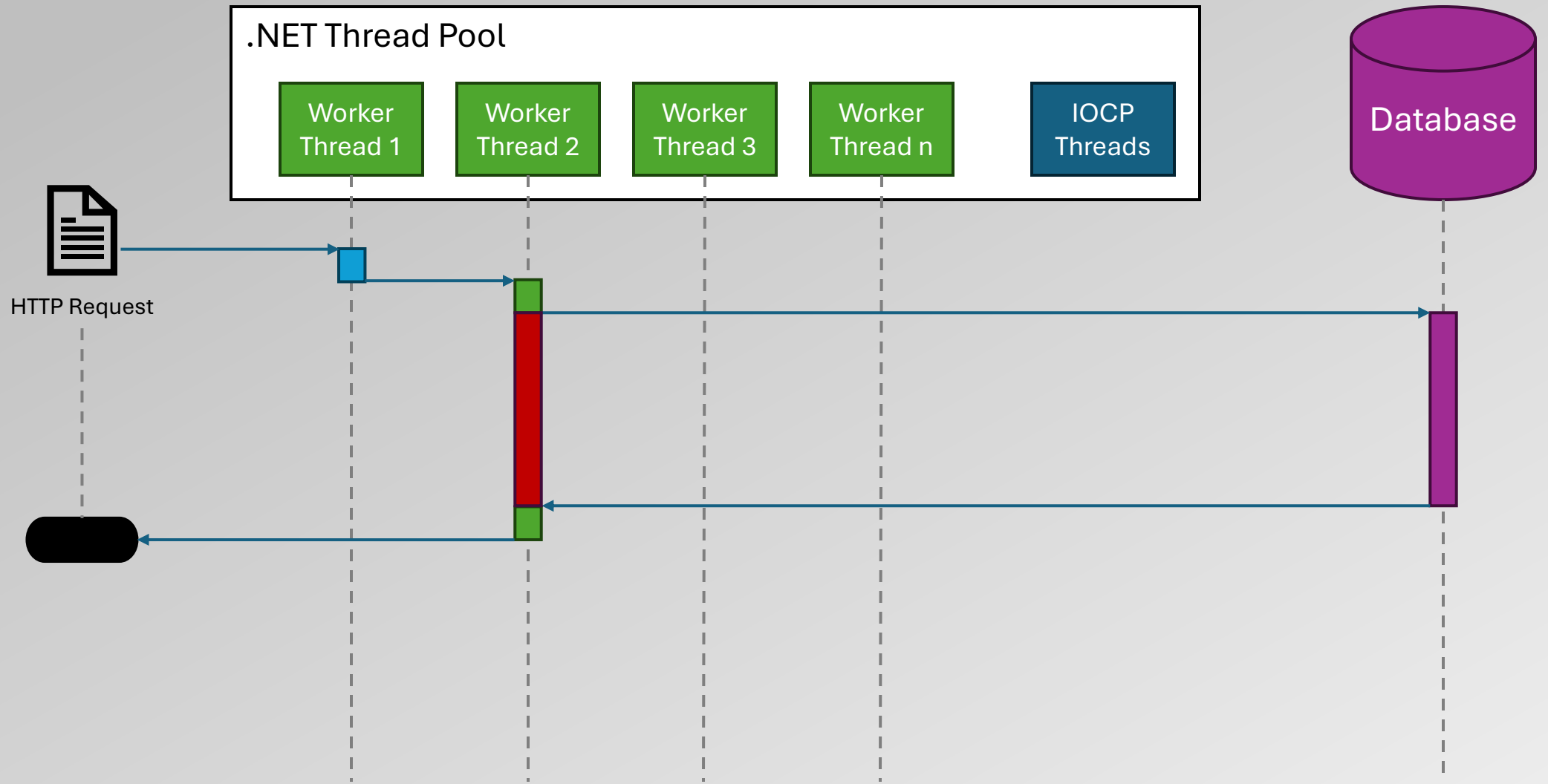
ASP.NET Core services are inherently multi-threaded



After await, you are probably on another Worker Thread

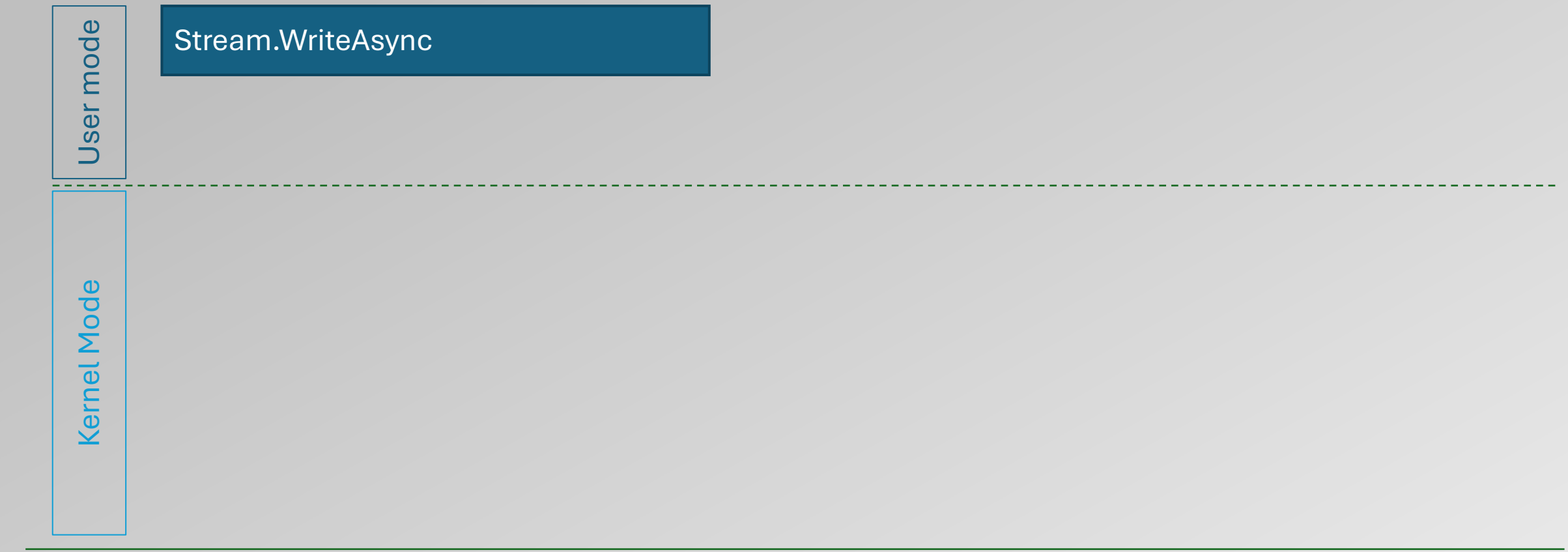
- When using async await, avoid **ThreadStatic**, **ThreadLocal<T>** or similar constructs unless you have a thread with a **SynchronizationContext** (not present in ASP.NET Core, but in classic ASP.NET).
- If you absolutely need to, then use **AsyncLocal<T>** (if you know what you are doing), but prefer to simply use scoped dependencies in ASP.NET Core.
- The **ExecutionContext** will automatically capture the state of **AsyncLocal<T>** before an async operation starts and restores this state when the corresponding continuation is run.
- This way, side-effects like impersonation won't leak to other methods that are running on the same thread that started an async operation.

What happens during synchronous I/O?

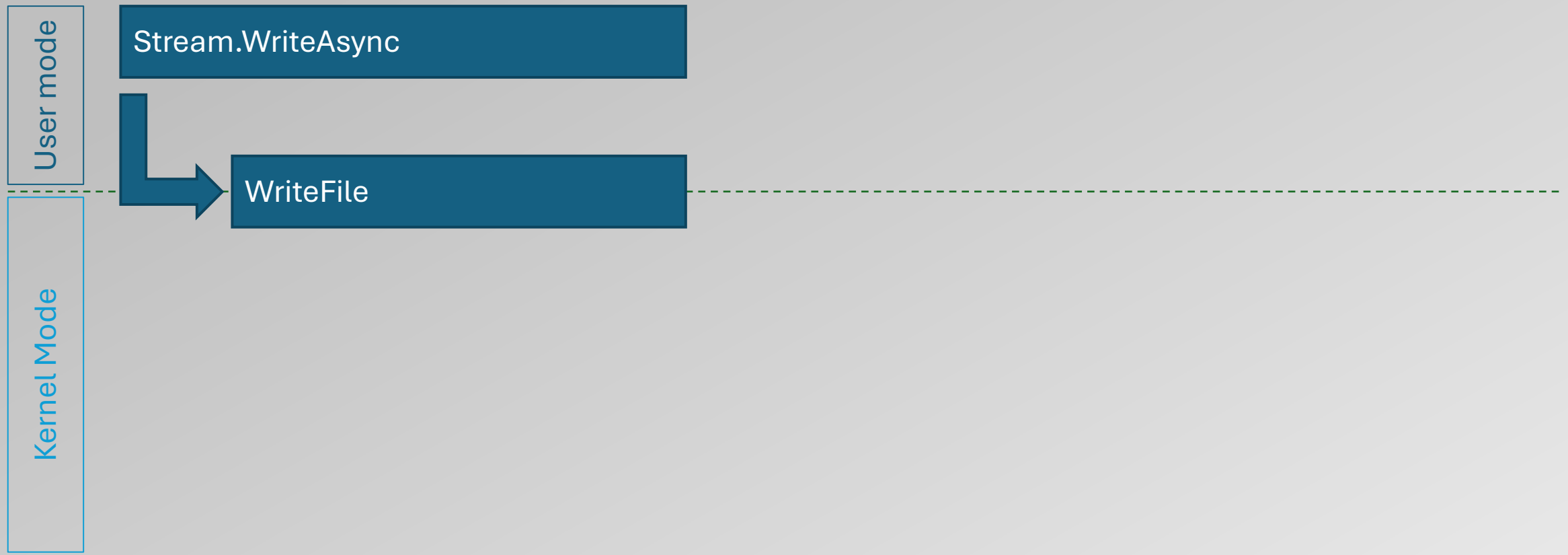


ppedv

Async I/O Write File operation on Windows



Async I/O Write File operation on Windows

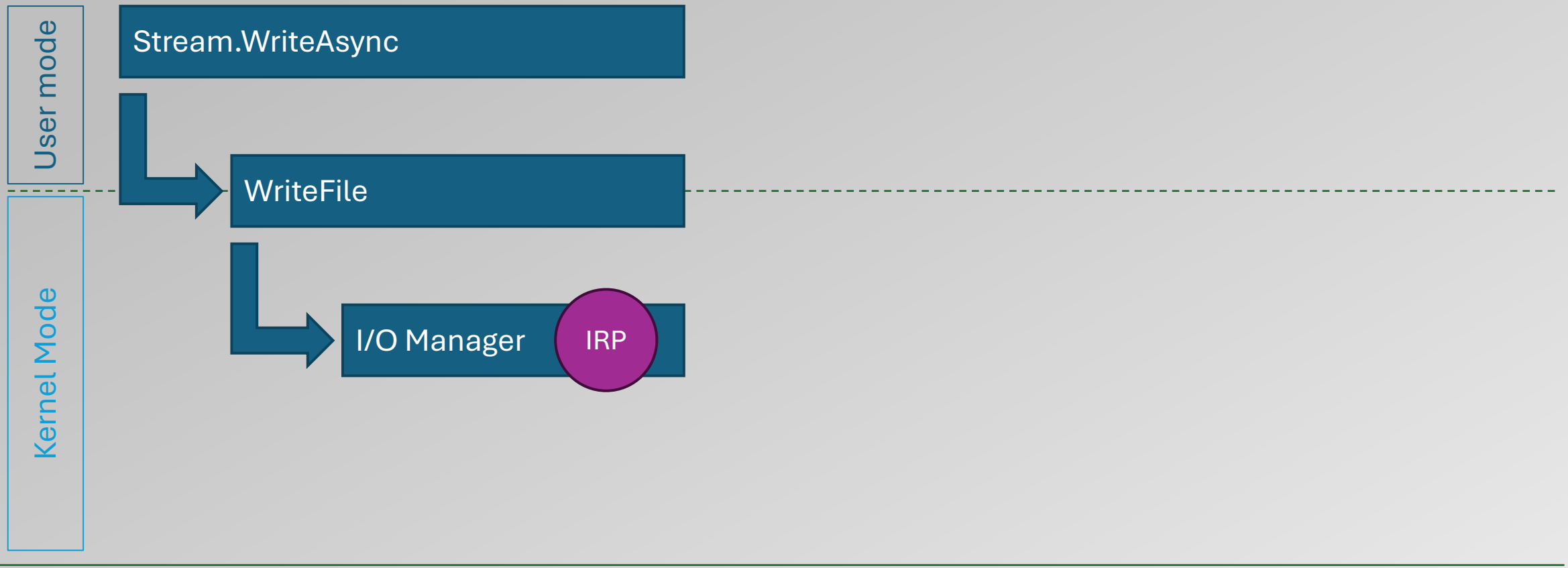


ppedv

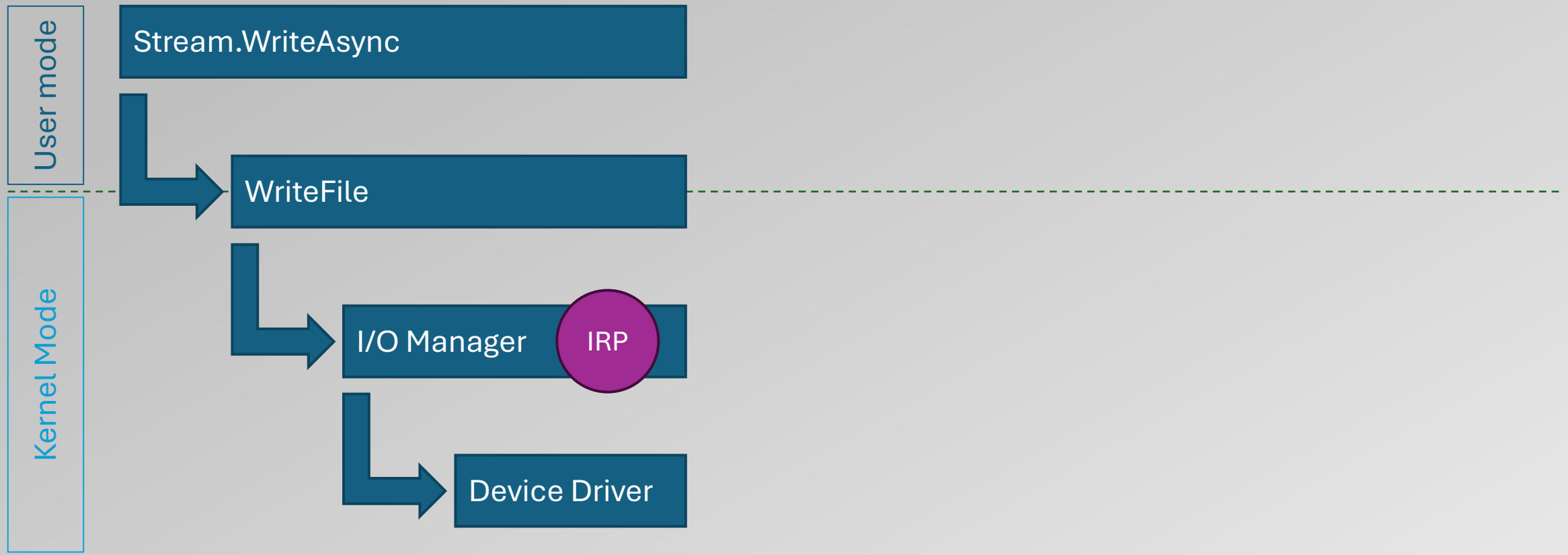
Async I/O Write File operation on Windows

ppedv

Async I/O Write File operation on Windows

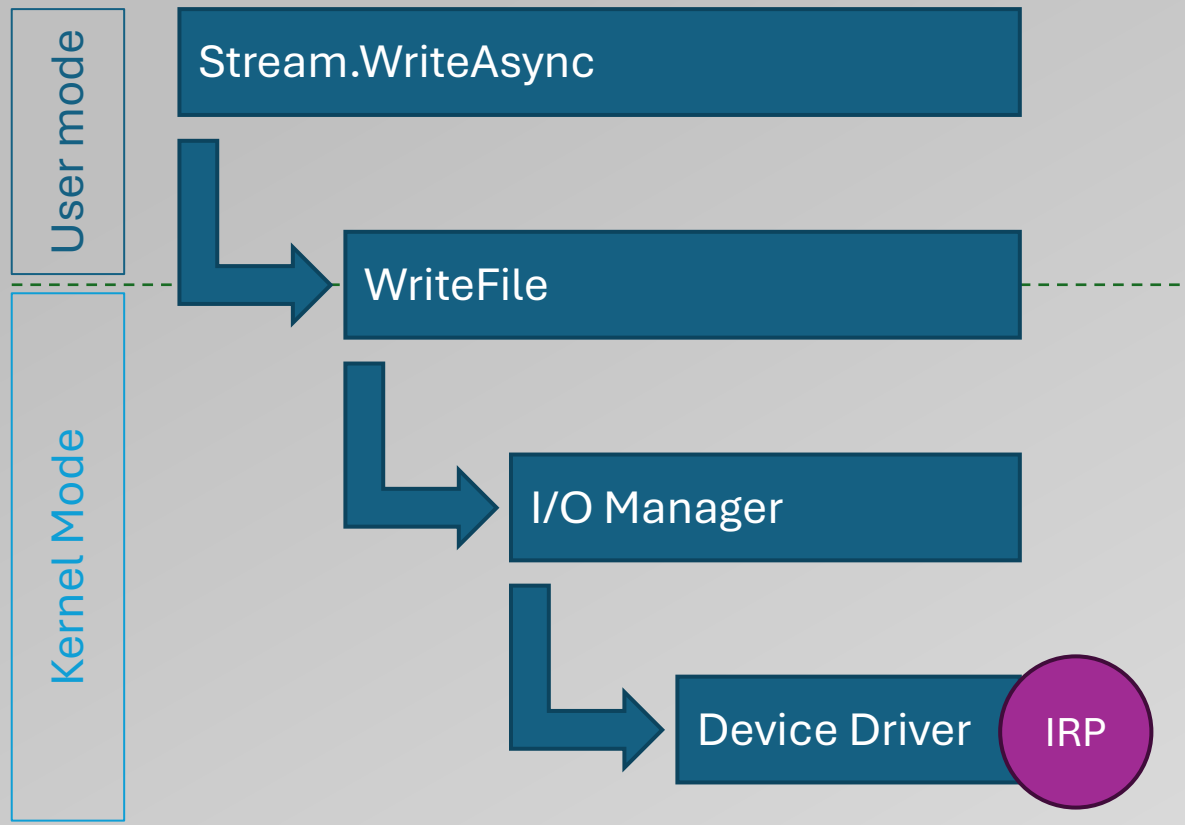


Async I/O Write File operation on Windows

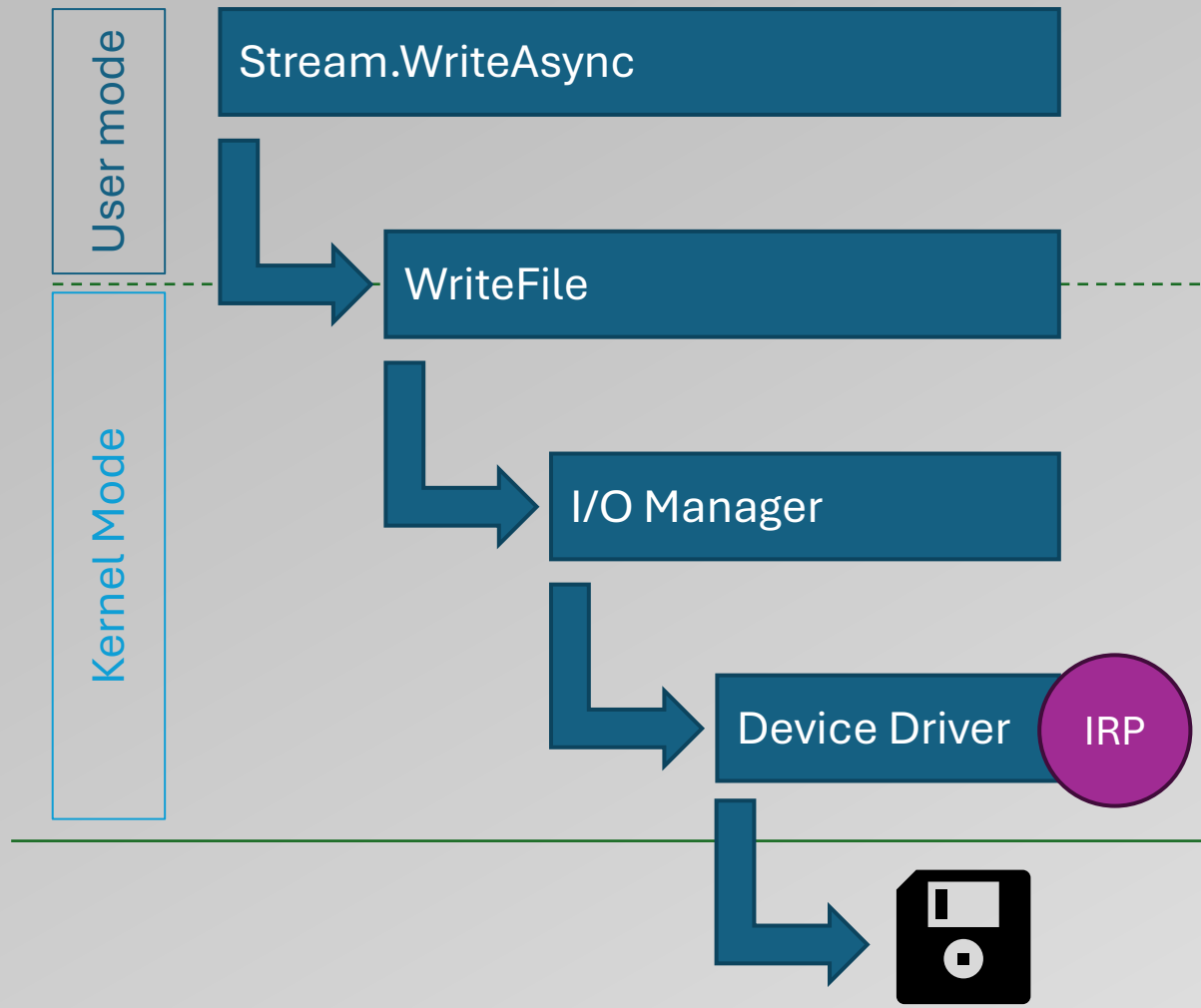


ppedv

Async I/O Write File operation on Windows

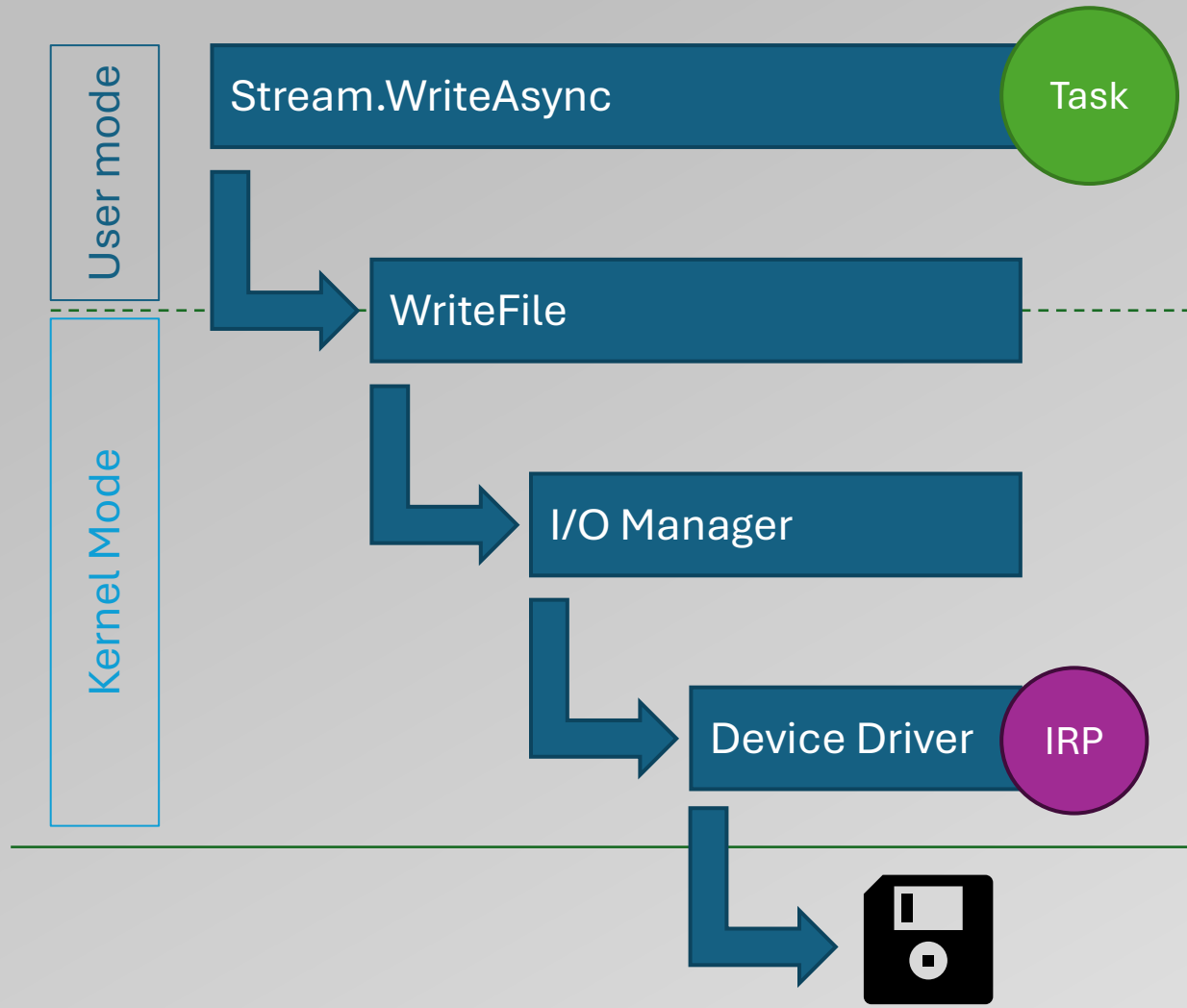


Async I/O Write File operation on Windows



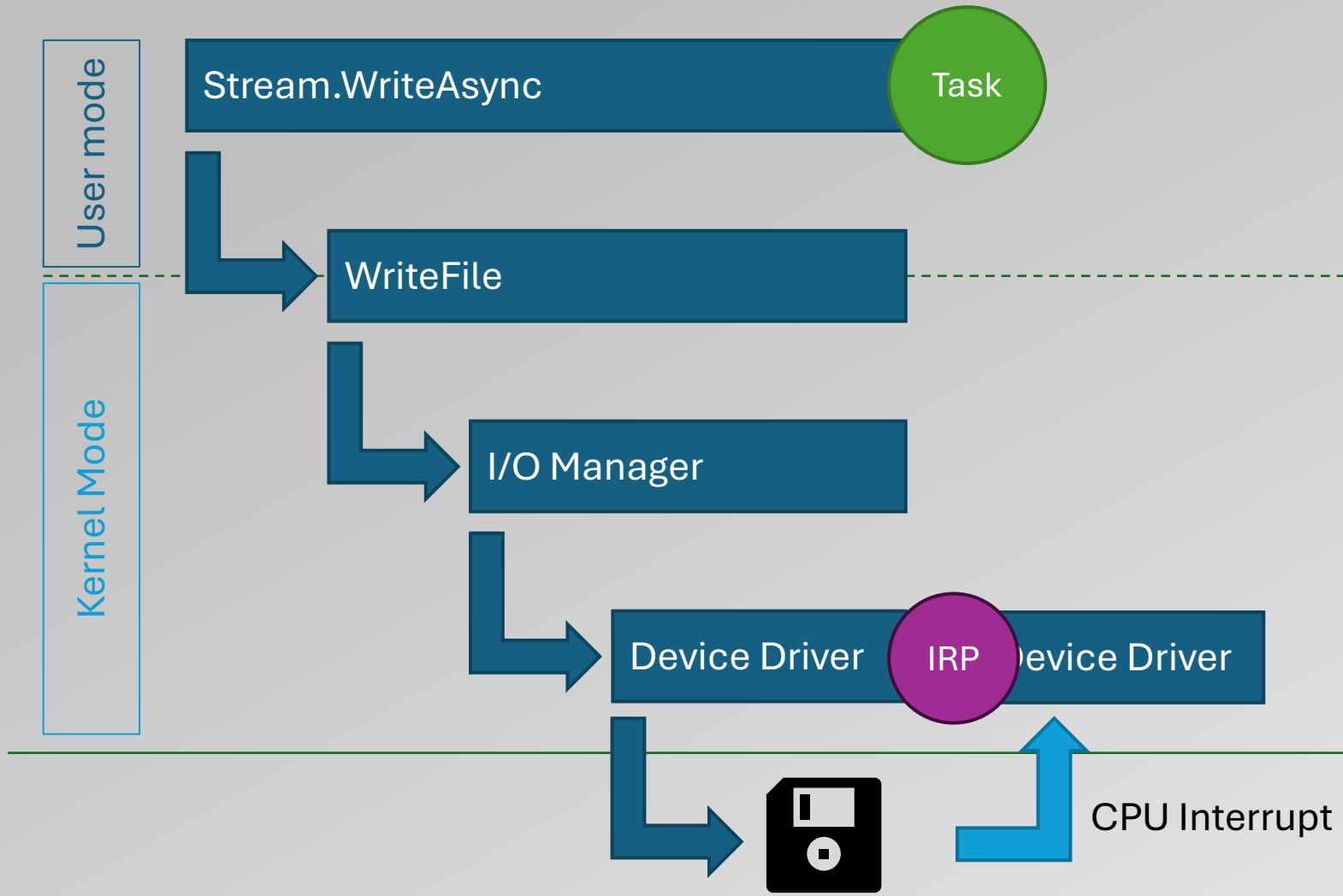
ppedv

Async I/O Write File operation on Windows

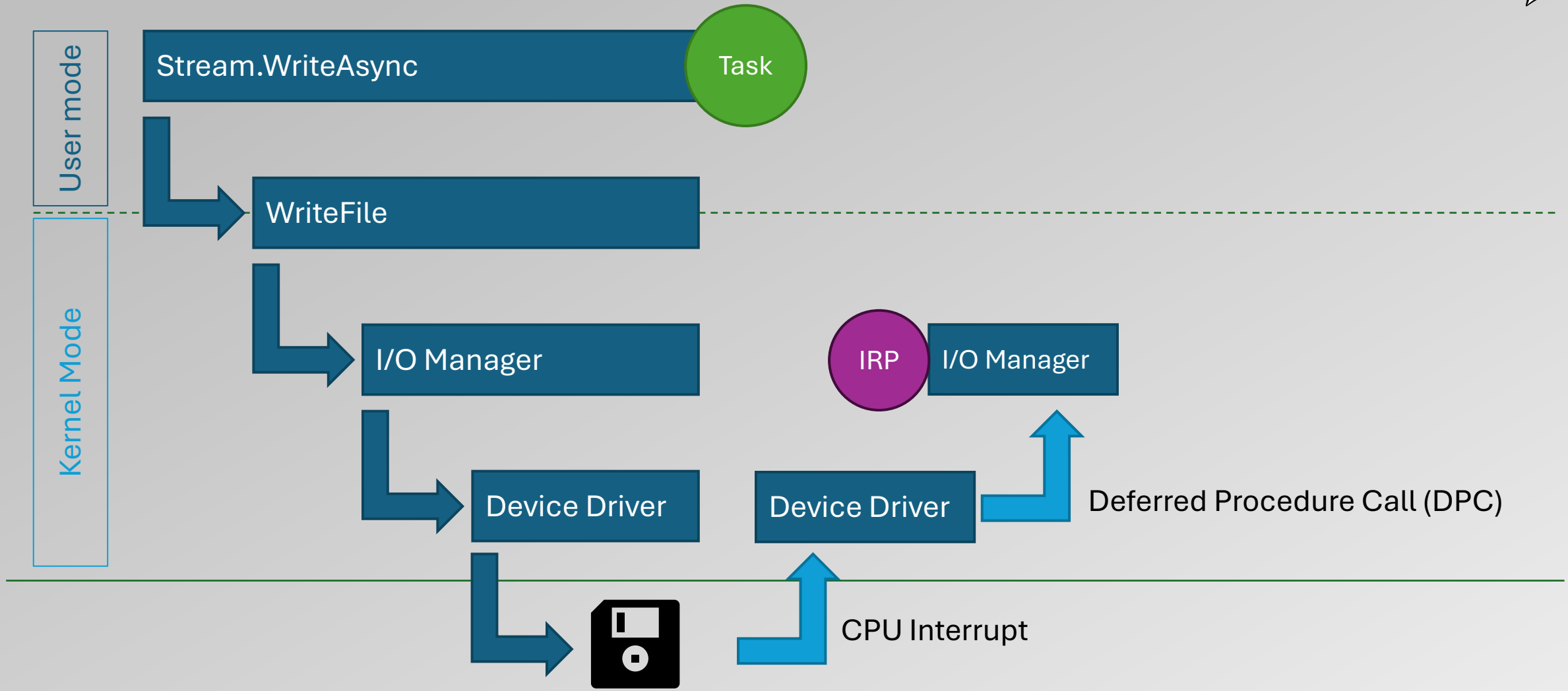


ppedv

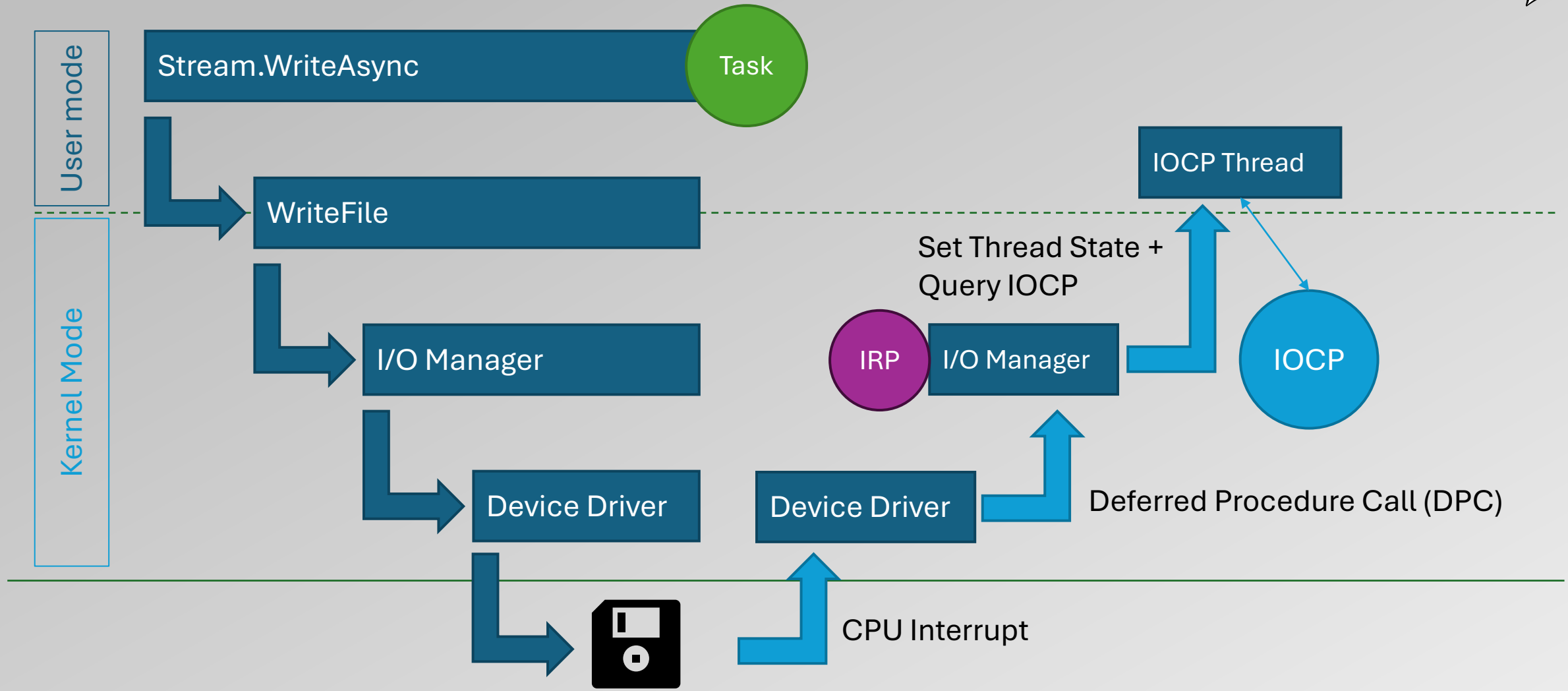
Async I/O Write File operation on Windows



Async I/O Write File operation on Windows

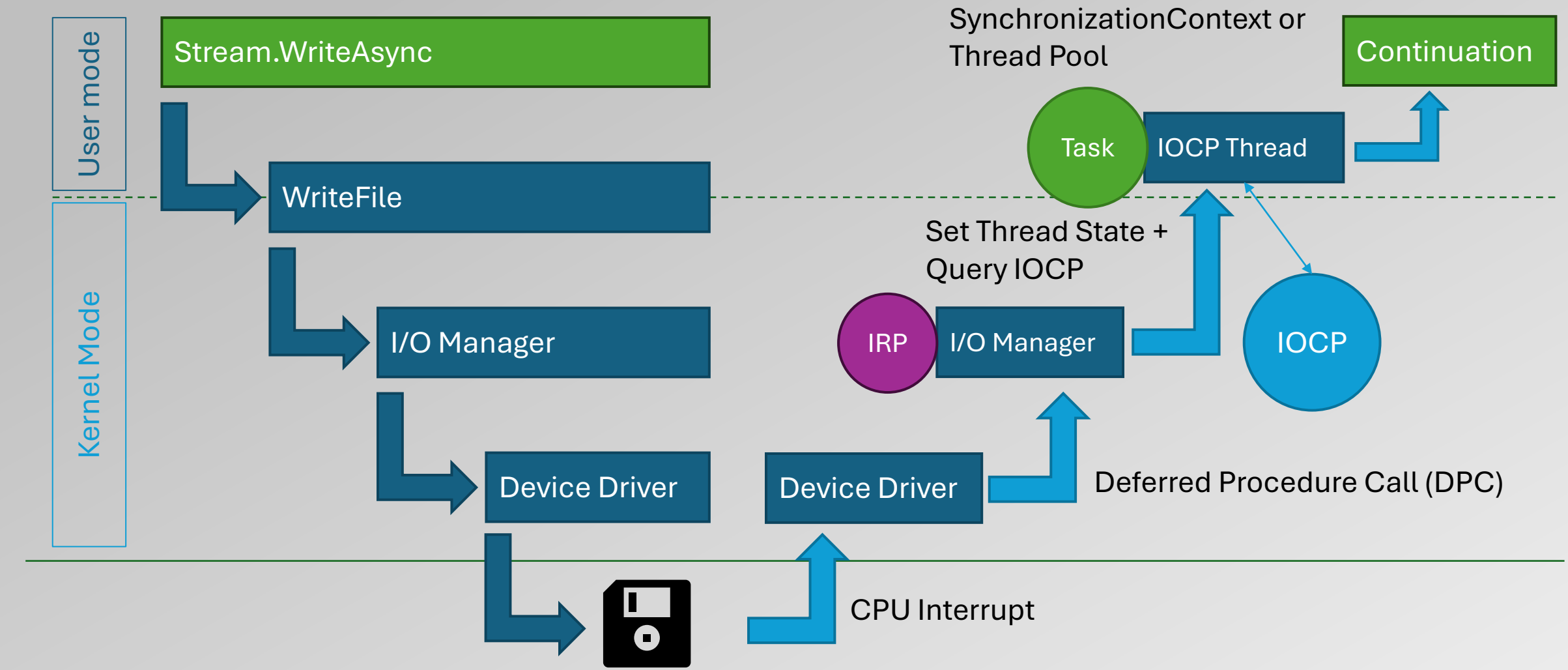


Async I/O Write File operation on Windows



ppedv

Async I/O Write File operation on Windows



ppedv

What are these IOCP threads of the Thread Pool?

- IOCP stands for I/O Completion Port and is the central mechanism of the Windows I/O manager for notifying callers about async I/O events.
- IOCP threads of the .NET Thread Pool bind to one IOCP and then block.
- You can bind as many threads as you want to one IOCP port.
- If an IOCP event is ready, the Windows I/O manager will choose one bound thread and set its state to "Ready to run", so that it can dequeue the event from the IOCP.
- After dequeuing, the IOCP thread will update the corresponding task and enqueue the continuation either on the Thread Pool or on the original caller's **SynchronizationContext**.
- If you run on Linux or MacOS X, there are similar mechanisms like epoll and kqueue.

How do we avoid synchronous I/O?

```
dbContext
    .Posts
    .Where(p => p.PublishDate >= from &&
                p.PublishDate <= to)
    .OrderBy(p => p.PublishDate)
    .ToList();
```

```
dbContext
    .Posts
    .Where(p => p.PublishDate >= from &&
                p.PublishDate <= to)
    .OrderBy(p => p.PublishDate)
    .ToListAsync();
```

But only if your
data access library plays along...

ppedv

Targeting different third-party systems with async I/O

- HttpClient, GRPc services, and Sockets are async by default.
- FileStream has async support since .NET 6 without performance degradation.
- ADO.NET abstractions have async methods, but their default implementation executes synchronously (providers must override these methods and implement proper async I/O).
- Object/Relational Mappers (ORMs) usually build on top of ADO.NET - they won't fix the problems in underlying providers.
- What about SAP, ERP systems, or other third-party systems with proprietary protocols?

Does your third-party access library support async I/O?

The situation with third-party access libraries

- System.Data.SqlClient and Microsoft.Data.SqlClient support async I/O except transactions (might be coming to Microsoft.Data.SqlClient)
- Npgsql: full async support
- RavenDB.Client and MongoDB.Driver: full async support
- MySql.Data: no support at all, you should use MySqlConnector instead
- Oracle.ManagedDataAccess.Core: async support since version 23.4 (which was published in May 2024, please consider an update)
- SQLite: no async support (but usually only local files)
- RabbitMQ.Client: async support since version 7 (released on 2024-11-06)
- SAP Connector for .NET: no async support

One thing before we move on:
Be wary of mixing async I/O and async compute

```
await Task.Run(() =>
    dbContext
        .Posts
        .Where(p => p.PublishDate >= from &&
                    p.PublishDate <= to)
        .OrderBy(p => p.PublishDate)
        .ToListAsync()
);
```

```
await dbContext
    .Posts
    .Where(p => p.PublishDate >= from &&
                p.PublishDate <= to)
    .OrderBy(p => p.PublishDate)
    .ToListAsync();
```

What we learned in Part 1

- Do not block the worker threads of the Thread Pool.
- await will usually return to the caller, so that something different can be performed while an async operation is ongoing.
- Use an “Async” suffix when a method returns a task (unless for endpoints or test methods).
- Asynchronous methods can be implemented in a synchronous way – the opposite is not true.
- You might call async methods, but your data access library might still perform sync calls and block threads under the covers.

Part 2:
The async state machine

Hands on!

async await decompiled

async methods are transformed

- The C# compiler will transform any method that is marked with the async modifier.
- The resulting state machine has a MoveNext method that consists of your actual code and generated code for await expressions (depends heavily on Control Flow).
- In Debug mode, the state machine is a class instead of a struct.
- The fields of this struct consist of your original method's parameters, variables, the "this" reference (if the source method was not static), task awaiters, and a method builder (the reusable part of the state machine).
- MoveNext will likely return to the caller before the task completes.
- MoveNext uses labels and goto statements to jump to the correct continuation for the previous task.
- You can analyze your own methods by using an IL Viewer (lowered C# code).

async await - what you need to be aware of

- async await has a performance overhead: initializing the state machine, boxing and moving it to the Managed Heap, etc. -> distinguish between sync and async calls!
- Be aware that each async method gets its own state machine when it is called – generally, avoid nesting to many calls of async methods.
- If the last statement in your async method is returning a task, you can skip the async keyword to reduce the number of state machines created (useful for Humble Objects).
- Be aware what the last statement is (e.g. using keyword for disposal).
- async await has a drastic impact on our Object-Oriented design: asynchronous method signatures are usually required across the whole call chain, avoid sync-over-async scenarios (but sometimes, this is useful).

Hands on!

Write your own state machine

Write your own state machine

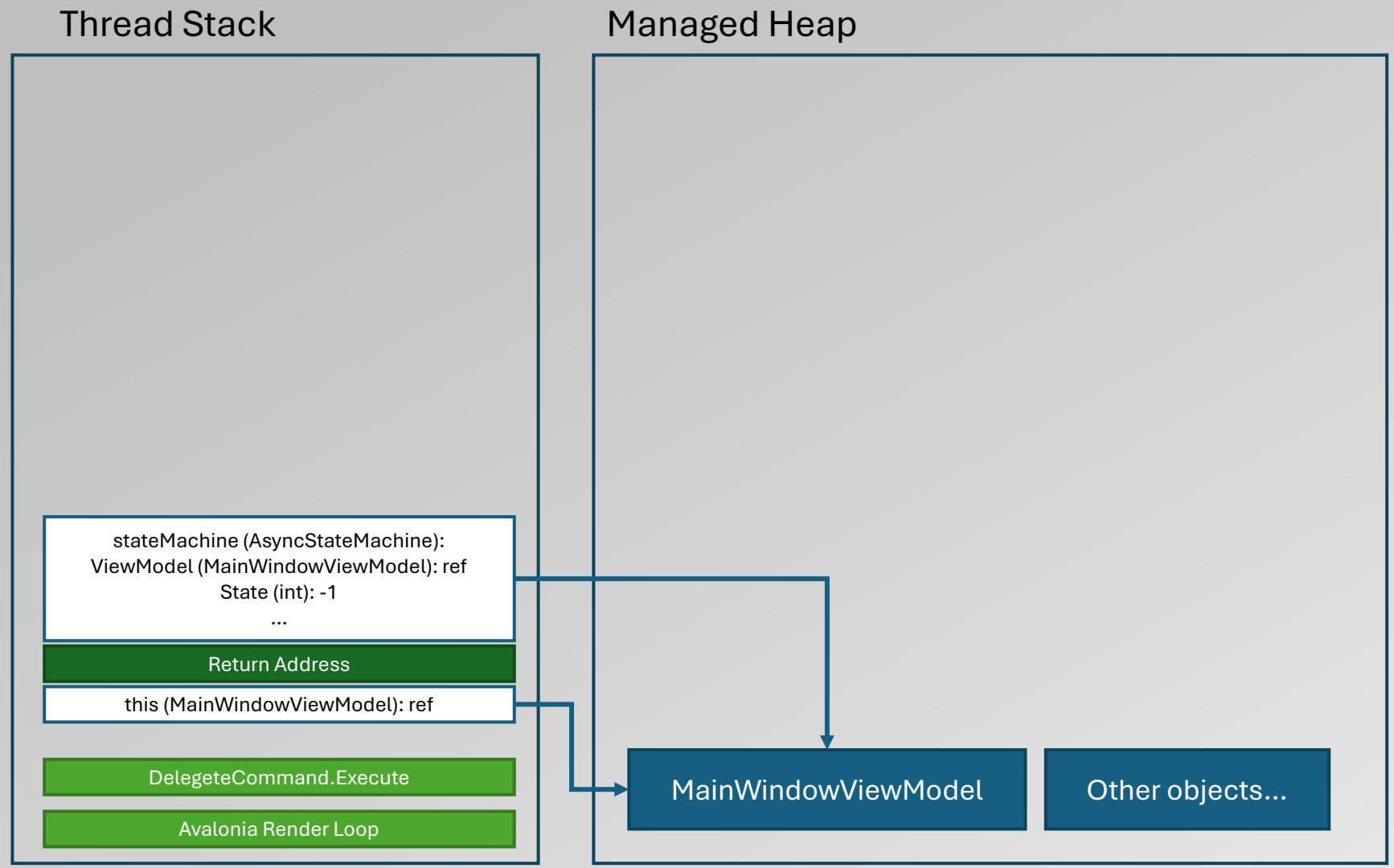
Please write your own state machine for the **DeleteContactEnpdoint.DeleteContact** async method

What happens in memory when an async method is called?

The memory management basics:

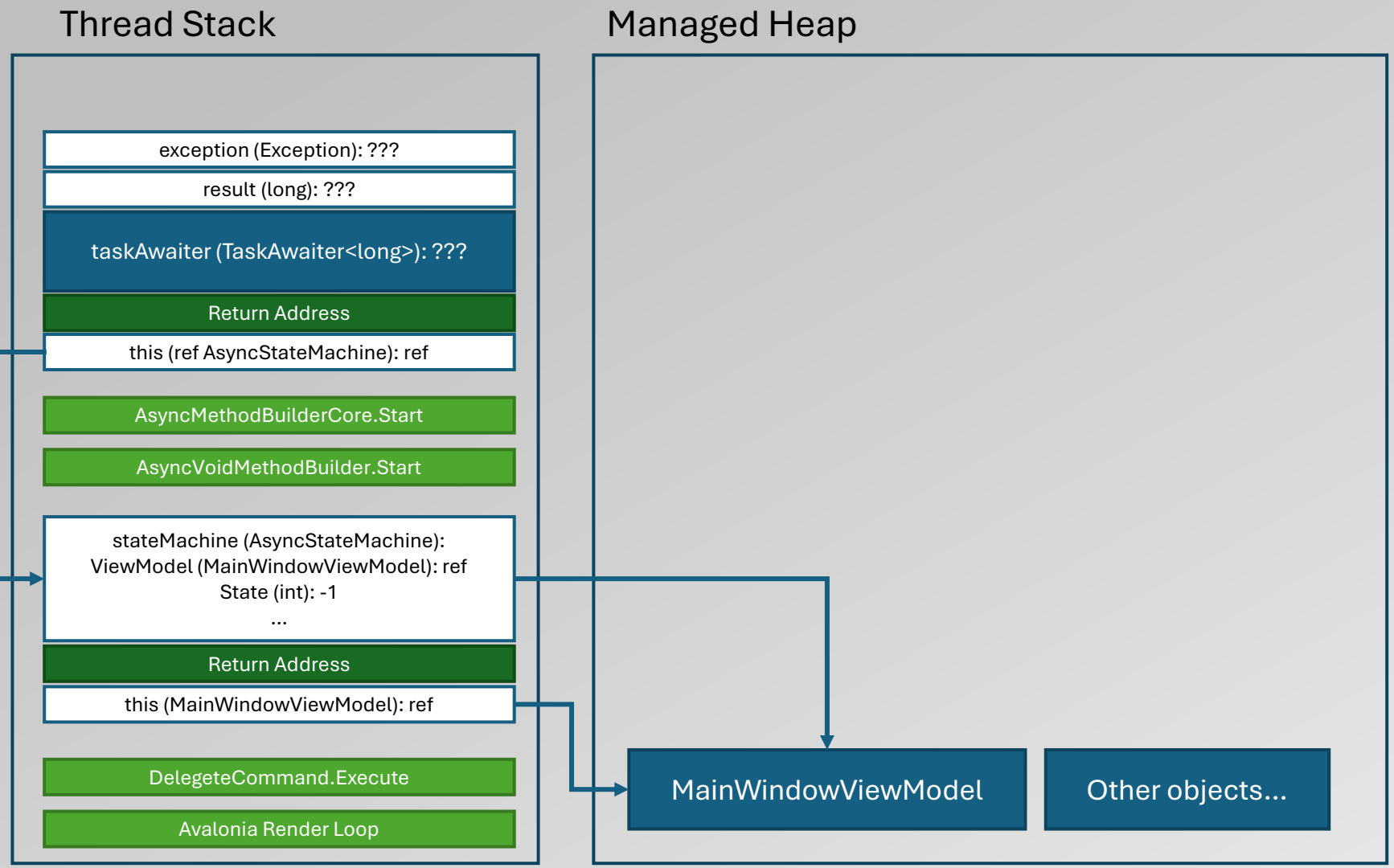
- We distinguish between thread stacks for parameters, variables, and return addresses vs. the managed heap (for objects).
- When a method is called, an activation frame is pushed on the thread stack.
- Each activation frame consists of parameters, return address, variables.
- When a method returns, its activation frame will be deallocated (by simply removing pointers).
- Value Types reside directly in a parameter/variable, or as part of an object. They can be boxed when addressed as a Reference Type!
- Reference types always point to an object in the heap.

Memory Snapshots: Before calling stateMachine.Builder.Start

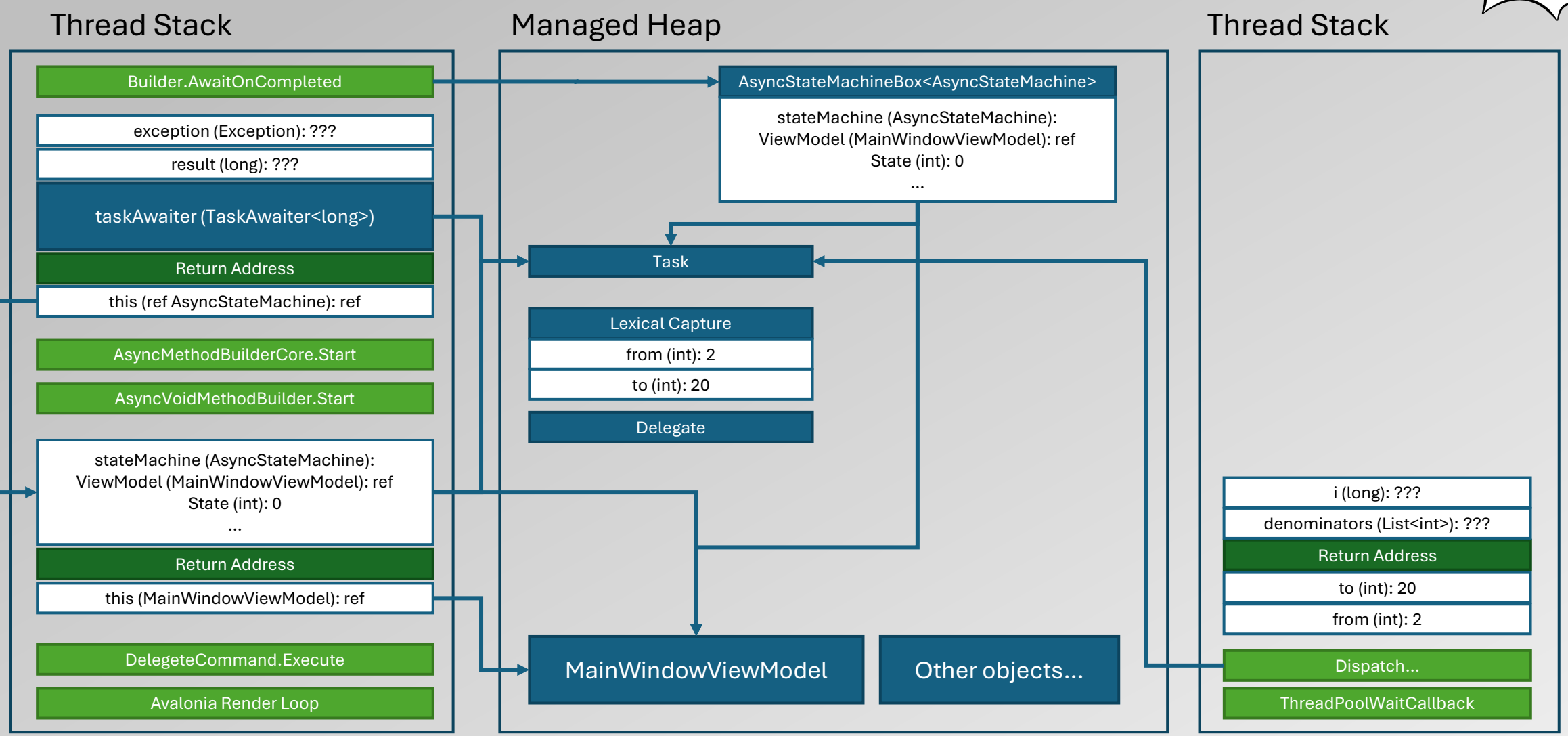


ppedv

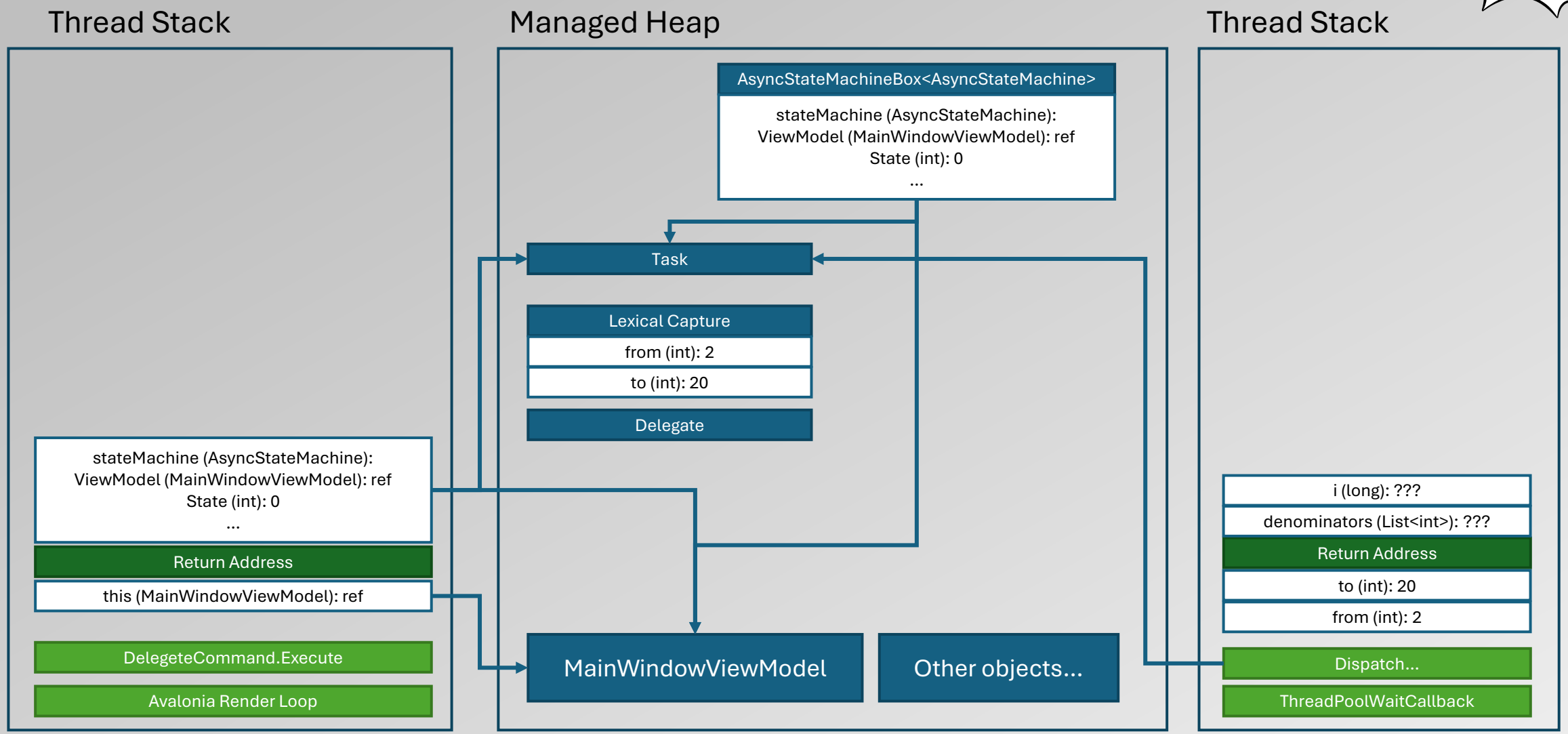
Memory Snapshots: At the beginning of MoveNext



Memory Snapshots: at the end of Builder.AwaitOnCompleted

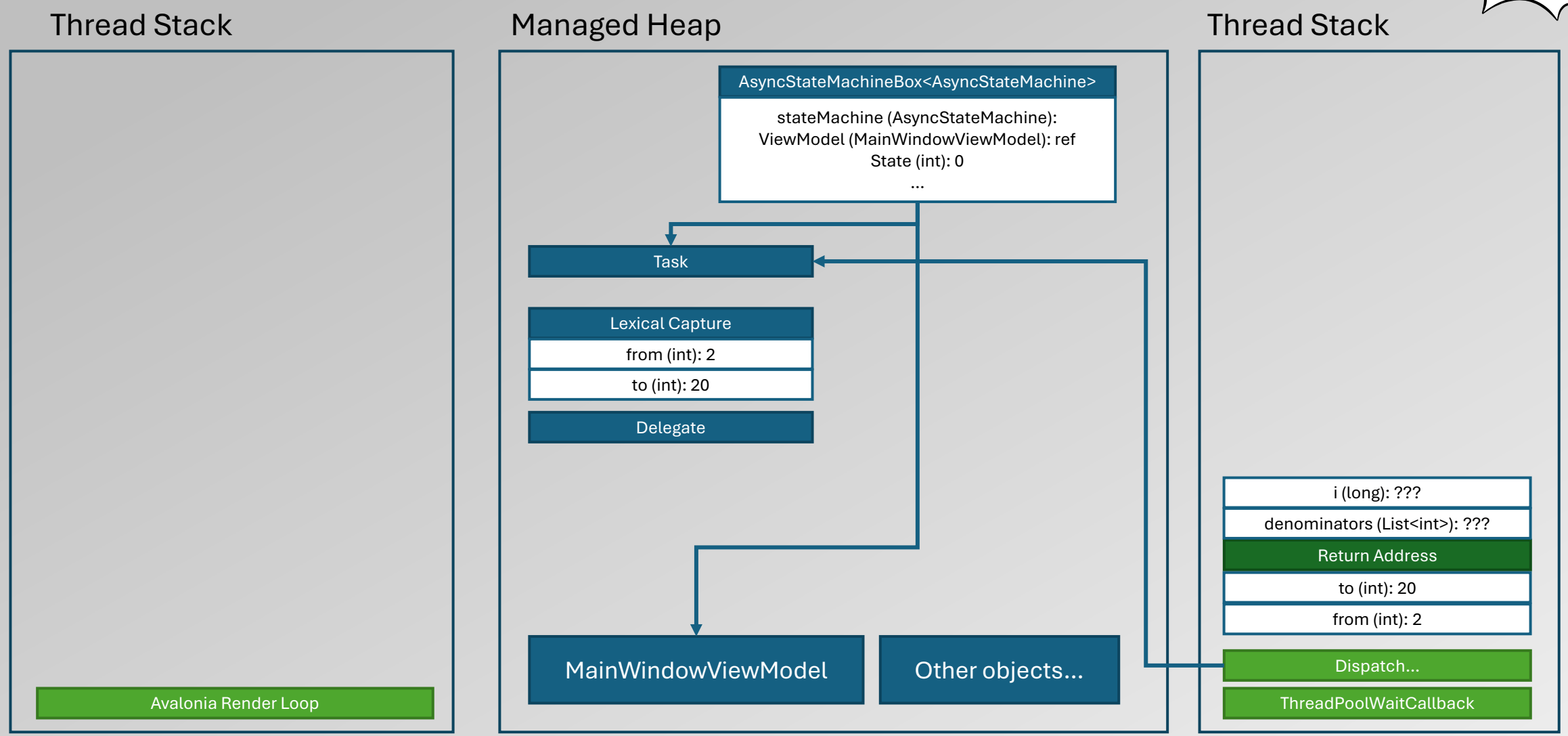


Memory Snapshots: return to calling method



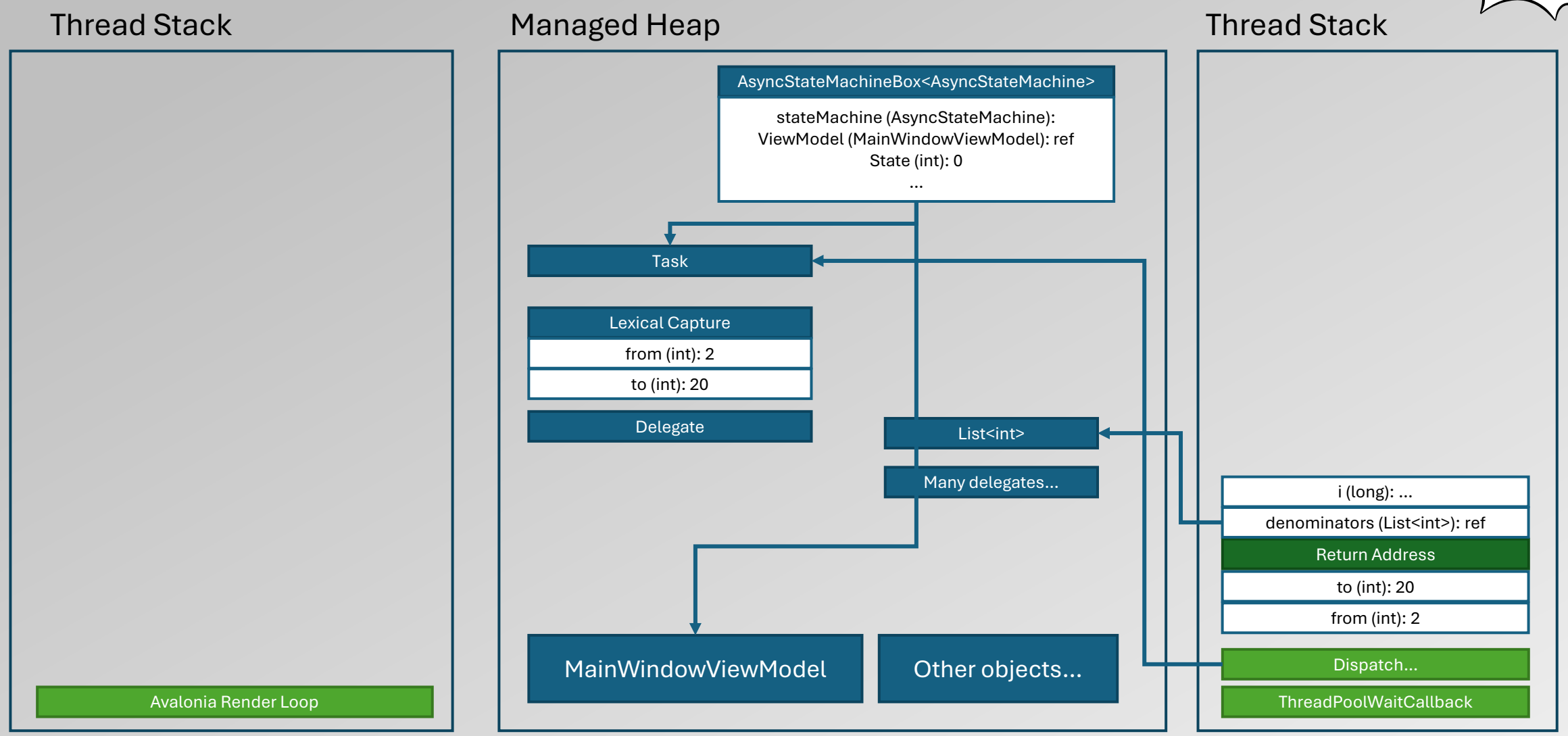
ppedv

Memory Snapshots: return to Avalonia Render Loop

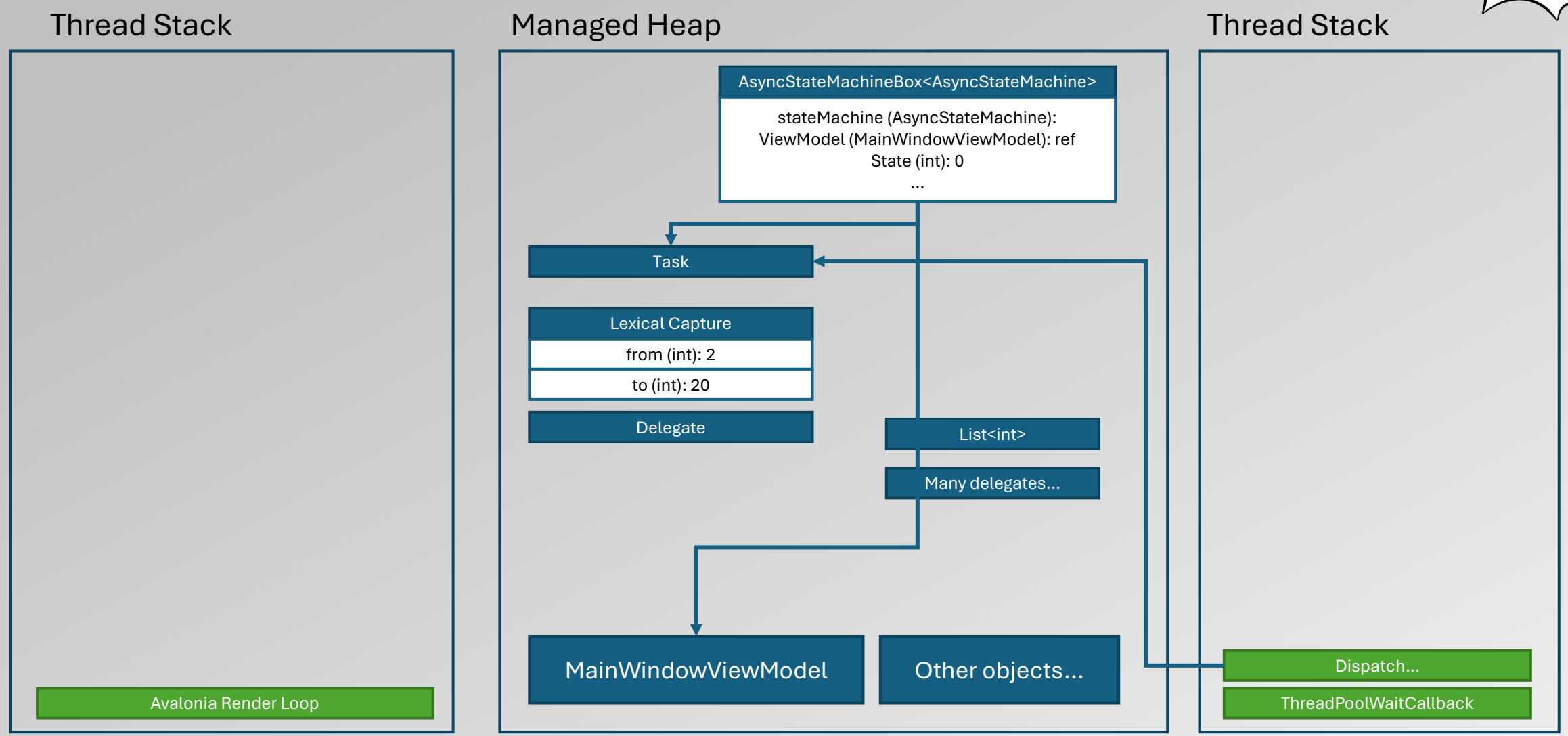


ppedv

Memory Snapshots: progress on background thread

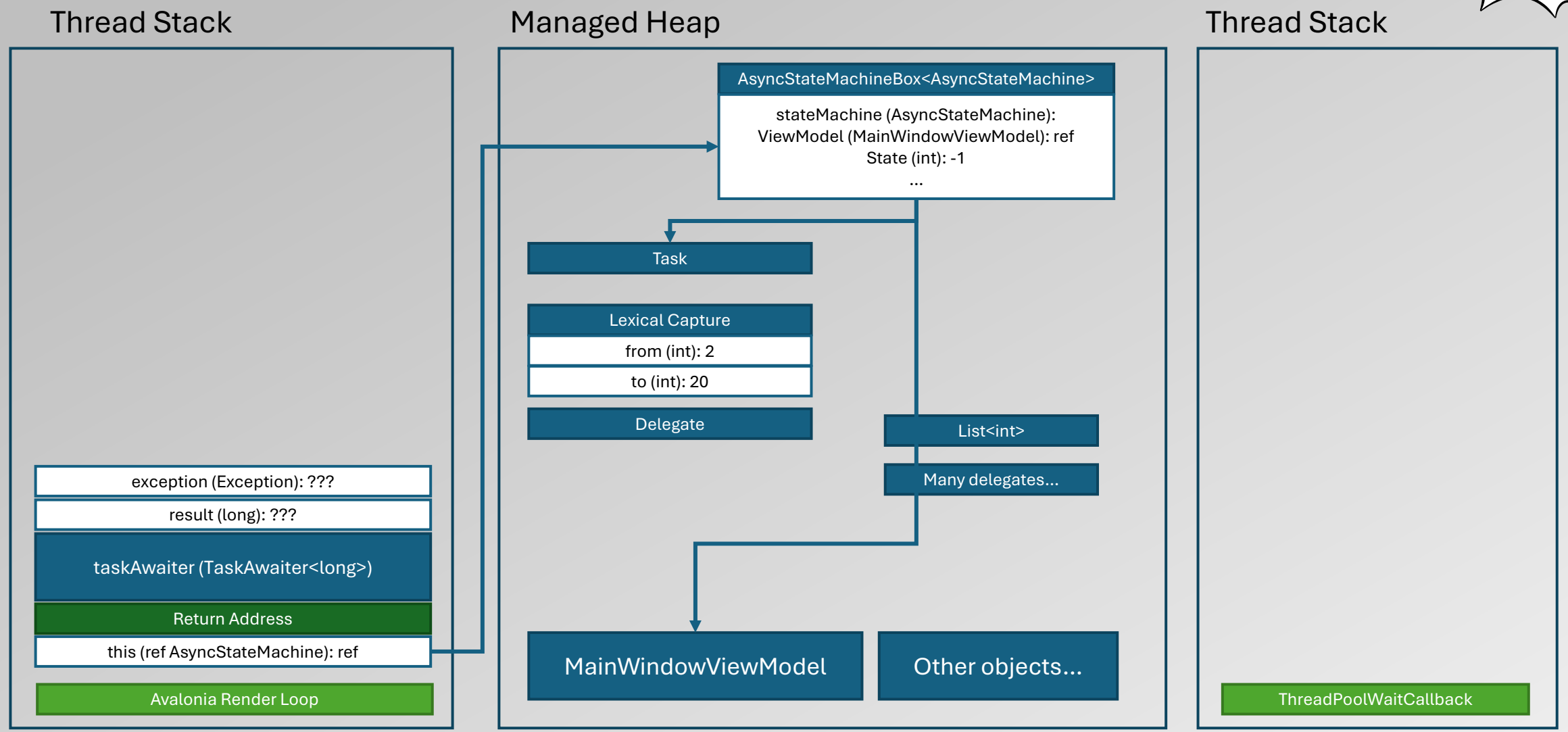


Memory Snapshots: after task is completed



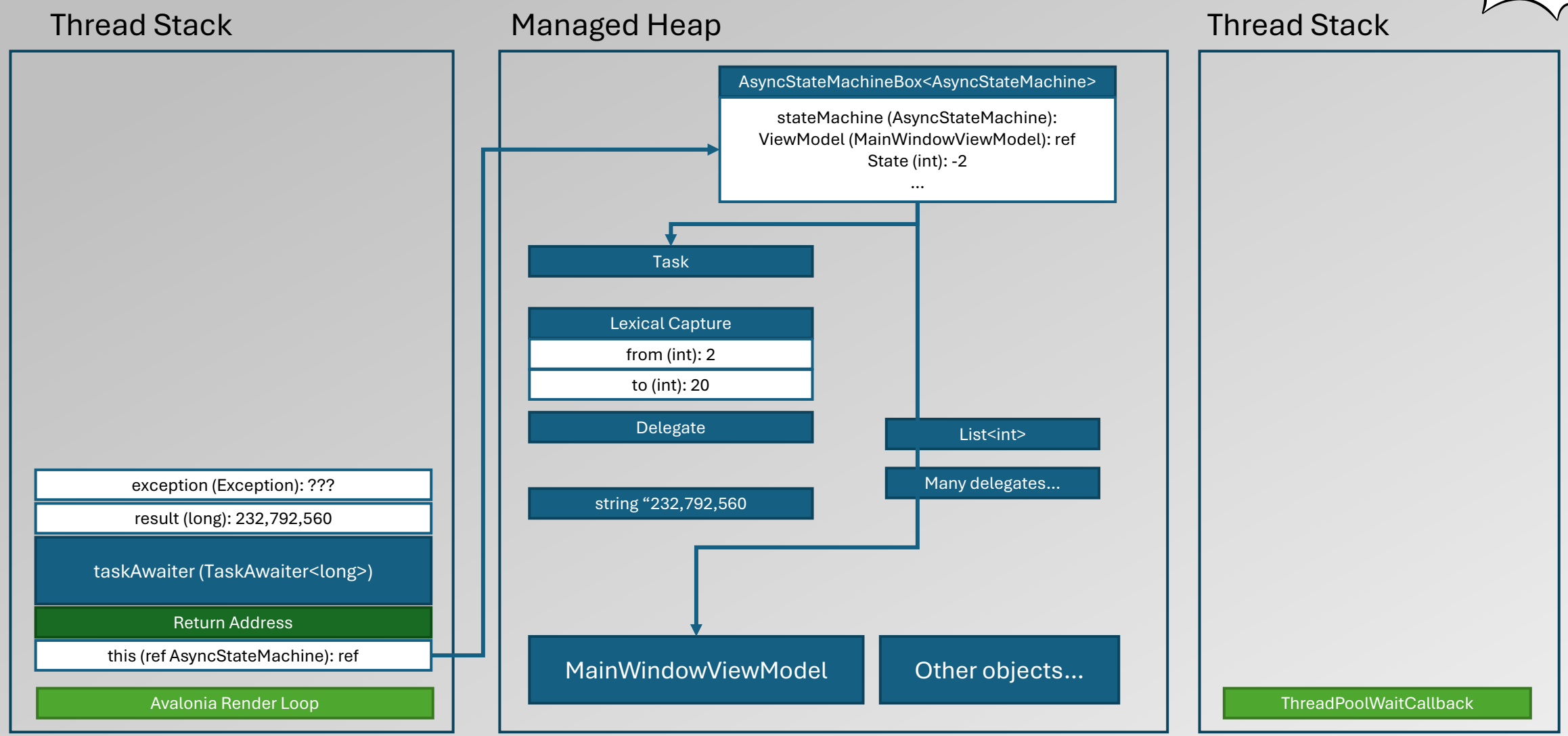
ppedv

Memory Snapshots: continuation started on UI thread



ppedv

Memory Snapshots: at the end of MoveNext



What can we learn from all this?

- Async methods are rewritten, parameters and variables become fields in a state machine.
- The state machine can return early while the async operation is still ongoing.
- This is why ref/out parameters cannot be used in async methods.
- When returning early, the state machine is boxed on the managed heap – this way, it survives when the activation frames of the initiating thread are released.
- When the async operation signals completion via the corresponding task, the async state machine's MoveNext method is executed on the original caller's thread (via SynchronizationContext) or on a worker thread of the .NET thread pool.

What about this benchmark?

ppedv

The issue with Nick's benchmark

- The async state machine overhead is not determined correctly: it always returns synchronously, which is why it will never be boxed and moved to the managed heap.
- To achieve that, he could have used **Task.Delay** – but then the performance benchmark gets unprecise because you have no guarantee when the continuation will be executed after the delay.
- This was rectified in a later video: https://youtu.be/aaC16Fv2zes

What have we learned so far?

- Each method which has the async modifier will be transformed into a state machine.
- The state machine has a **MoveNext** method which will be triggered in the beginning and for each continuation.
- The method builder is a reusable part of the state machine, able to start it and begin continuations.
- If **MoveNext** does not finish synchronously, the state machine is boxed and moved to the heap.
- After that, we return to the caller which can use the thread for other purposes (handle further HTTP requests for example).

Part 3: IAsyncEnumerable and CancellationTokens

IAsyncEnumerable

- Asynchronous variant of an enumerable.
- We also have the concept of an **enumerable** and an **enumerator** here.
- **MoveNextAsync** is the important part: here, the next element is obtained asynchronously (and you'll probably end up on another thread).
- The whole enumeration can be cancelled by cancellation token passed to **GetAsyncEnumerator**.

Creating IAsyncEnumerable instances with yield return

- Async methods that return **IAsyncEnumerable<T>** can use yield return (and await, of course).
- Cancellation tokens should be marked with the **EnumeratorCancellation** attribute.
- The underlying state machine will throw an **OperationCancelledException** on calls to **MoveNextAsync** when the token marked with EnumeratorCancellation is cancelled.

Consuming IAsyncEnumerable with await foreach

- Use **await foreach** to consume IAsyncEnumerable<T>.
- It is lowered in a similar fashion to a regular foreach loop.
- When a method producing an IAsyncEnumerable<T> has no CancellationToken parameter, you can still cancel the stream by using the WithCancellation extension method – this will throw an **OperationCancelledException** during **MoveNextAsync**.

The problem with IAsyncEnumerable<T> and yield return

- Each caller gets its own state machine.
- Sometimes, you want to have a single source and stream to multiple consumers.
- You can use System.Reactive's **IObservable<T>** to achieve this – there is a handy **ToAsyncEnumerable** extension method that registers a new observer which is triggered when the observable is updated (for example via a **Subject<T>**).

Cancellation Tokens

- Cancellation Tokens allow callers to cancel asynchronous operations.
- It depends on the implementation, but usually, a **TaskCancelledException** or its base class **OperationCancelledException** is thrown.
- You can use the **Register** method to execute a delegate when a cancellation token is cancelled – the returned CancellationTokenRegistration must be disposed by you.
- If you need to create a CancellationToken, use **CancellationTokenSource**. Be sure to dispose it.
- Frameworks like MVC, Minimal APIs, and GRPC offer a **CancellationToken** when your endpoint is called – forward them to your services to for example cancel a database operation when the request is cancelled by the caller.

Synchronization Primitives: lock

- The **lock** keyword is used to enter a so-called Critical Section: this is code block that only one thread can execute concurrently.
- While one thread is within the Critical Section, other threads block until the thread leaves the critical section.
- lock is lowered to a try-finally block and uses **Monitor.Enter** to acquire entrance to the critical section, and **Monitor.Exit** to exit the critical section (Lock in .NET 9 is more performant).
- **await** cannot be used within locks: as await returns to the caller (unless the operation finishes synchronously), the critical section would be left too soon.
- However, you can use **lock** in async methods as long as there is no **await** in its scope.

Demo Time!

Async GRPC Streaming
with Cancellation

What have we learned so far

- **IAsyncEnumerable<T>** is great for streaming data, even across process boundaries.
- You can cancel an **IAsyncEnumerable<T>**, even if the method creating it accepts no Cancellation Token.
- MVC, Minimal APIs, and GRPC offer a Cancellation Token for each request.
- The **lock** keyword can be used with asynchronous methods, just do not use **await** within its scope.

Part 4:
Triggering long-running asynchronous operations

Transactional Outbox - Why do we need it?

- Common scenario in distributed systems: workflow step is finished, state change is persisted, afterwards and event is published.
- In some rare cases, the event gets lost when an exception is thrown during the call to Publish.

```
public static async Task<IResult> CompleteOrder(
    CompleteOrderDto dto,
    ICompleteOrderDbSession dbSession,
    IPublishEndpoint publishEndpoint,
    CancellationToken cancellationToken = default)
{
    var order = await dbSession.GetOrderAsync(dto.OrderId, cancellationToken);
    if (order is null)
    {
        return TypedResults.NotFound();
    }

    order.State = OrderState.Completed;
    await dbSession.SaveChangesAsync(cancellationToken);

    await publishEndpoint.Publish(new OrderCompleted(order), cancellationToken);
    return TypedResults.Ok();
}
```

Transactional Outbox - Why do we need it?

- Some might suggest that we should publish the event before we complete the transaction with the database.
- But what happens when an exception occurs during SaveChangesAsync?

This does not solve the problem!

```
public static async Task<IResult> CompleteOrder(
    CompleteOrderDto dto,
    ICompleteOrderDbSession dbSession,
    IPublishEndpoint publishEndpoint,
    CancellationToken cancellationToken = default)
{
    var order = await dbSession.GetOrderAsync(dto.OrderId, cancellationToken);
    if (order is null)
    {
        return TypedResults.NotFound();
    }

    order.State = OrderState.Completed;
    await publishEndpoint.Publish(new OrderCompleted(order), cancellationToken);
    await dbSession.SaveChangesAsync(cancellationToken);

    return TypedResults.Ok();
}
```

One solution: Saving state and message to the database

- We can save the domain state change as well as the message to the database within the same transaction.
- Either the transaction fails and nothing is saved, or both the domain state and the message is saved.
- A downstream component will load outbox items and send it to the Message Broker.

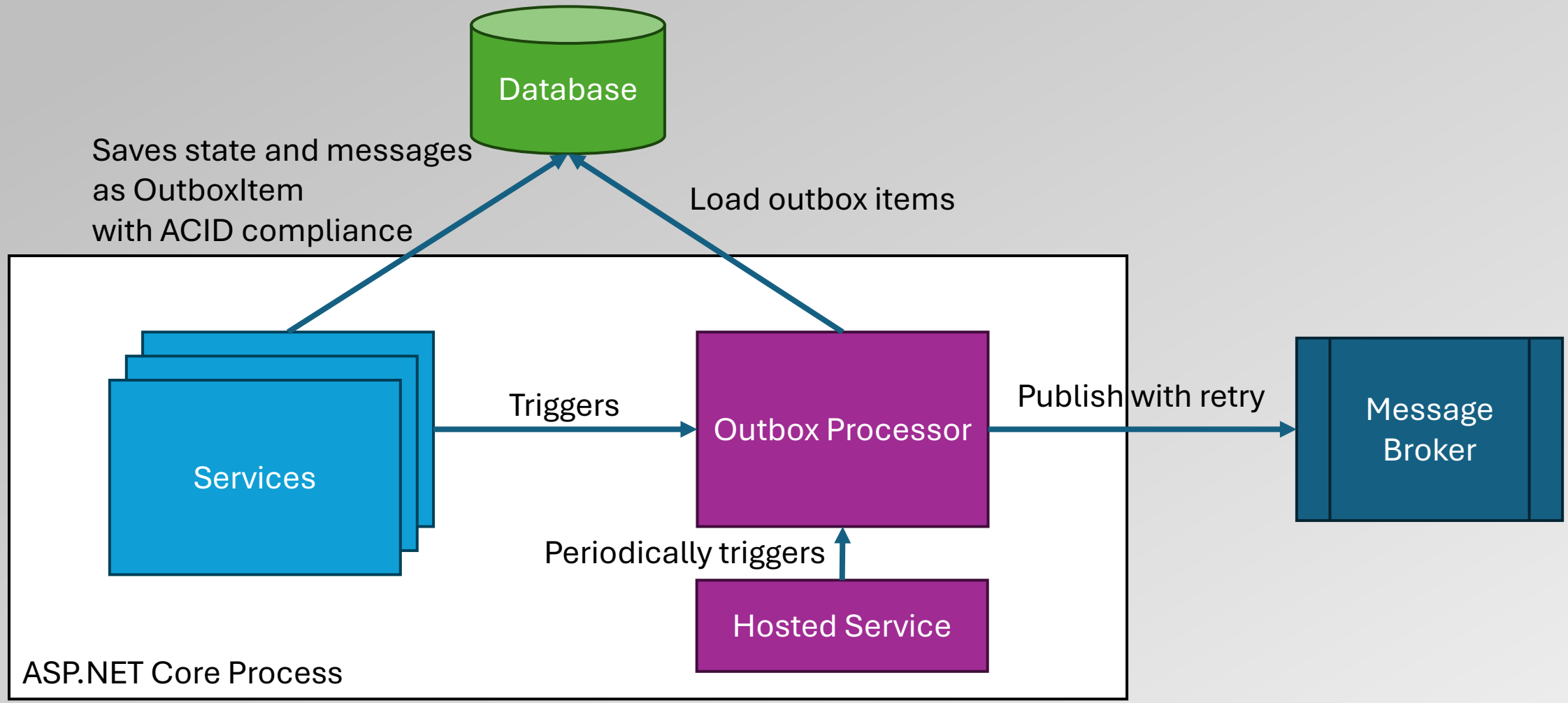
This way, we can ensure that the event is sent at least once.

```
public static async Task<IResult> CompleteOrder(
    CompleteOrderDto dto,
    ICompleteOrderDbSession dbSession,
    IPublishEndpoint publishEndpoint,
    CancellationToken cancellationToken = default)
{
    var order = await dbSession.GetOrderAsync(dto.OrderId, cancellationToken);
    if (order is null)
    {
        return TypedResults.NotFound();
    }

    order.State = OrderState.Completed;
    dbSession.AddMessageAsOutboxItem(new OrderCompleted(order));
    await dbSession.SaveChangesAsync(cancellationToken);

    return TypedResults.Ok();
}
```

Transactional Outbox Overview



ppedv

Requirements for the Outbox Processor

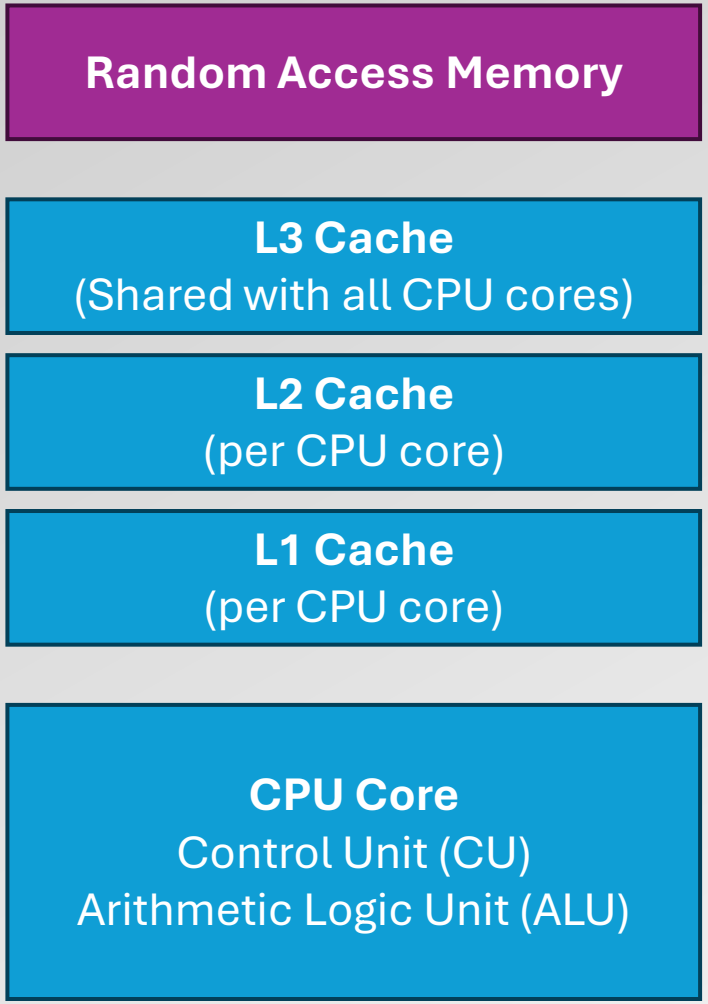
- The outbox processor must be implemented in thread-safe way because concurrent requests could trigger it.
- When the outbox processor is triggered by a service, it should take a maximum amount of about 50ms to enter the critical section.
- When the outbox processor is triggered, it should incorporate a Double-Check Lock to avoid waiting for the critical section in the first place.
- The trigger method should return if outbox processing was started or not by returning a Task<bool>.
- The trigger method should return as early as possible after starting outbox processing. The resulting processing task should not be awaited by the caller.
- Sending outbox items should be retried with a Polly ResiliencePipeline if necessary.

Synchronization Primitives: SemaphoreSlim

- The lock keyword won't cut it here. We cannot stop waiting for access into the critical section.
- Instead, we can use a **SemaphoreSlim** – its **WaitAsync** method allows us to asynchronously wait for entrance to the critical section.
- Avoid calling the Wait method, it blocks the calling thread while waiting for entrance to the critical section.
- Once acquired, we need to exit the semaphore with its Release method – use a try-finally block to accomplish this.
- Within the critical section, you can use await (this is different to the lock scope).
- A semaphore needs to be disposed.

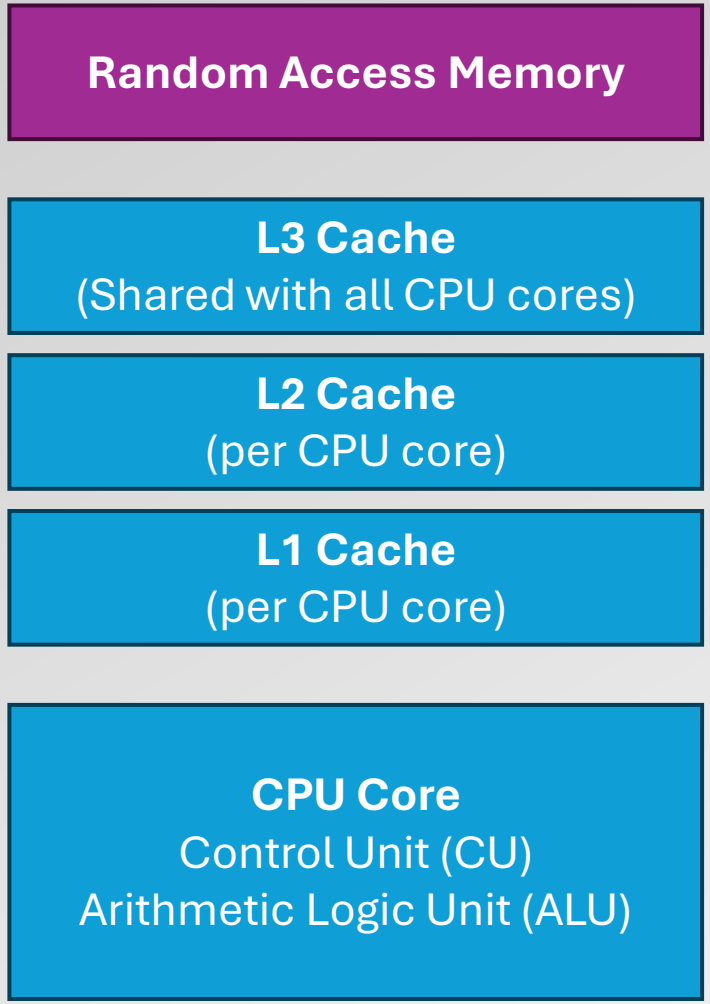
CPU Cache Optimization and Instruction Reordering

- CPU instruction sets (and therefore everything that builds on top of it) are by default optimized for single-threaded throughput
- Parameter, variable and even object field values can be kept inside CPU registers and the CPU Lx caches instead of being written back to memory (Cache Optimization)
- Instructions send to a CPU are processed by the Control Unit to utilize the ALU optimally – this can lead to Instruction Reordering (as long as the single-threaded post-conditions/invariants are kept intact)



Synchronization Primitives: Memory Barriers

- **Volatile.Read**, **Volatile.Write** (Half Fences) and **Thread.MemoryBarrier** (Full fences) solve the problem of Cache Optimization and Instruction Reordering.
- A full fence prevents read or write operations before it being reordered to after the fence and vice versa:
- Half-fences:
  - Volatile-Read: read or write operations after the fence cannot be moved before it.
  - Volatile-Write: read or write operations before the fence cannot be moved after it.
- All fences synchronize a value so that it is visible for other cores (Cache Coherency Protocol, MESI).



Memory Barriers are included in other constructs

The following things have an explicit full fence at the beginning and at the end:

- await
- lock, Monitor.Enter, Monitor.Exit
- Interlocked, Thread
- asynchronous callbacks and task continuations
- All Signals (AutoResetEvent, ManualResetEvent, etc.)

Double-Check Locks (1)

```
if (_currentTask is not null)
    return;

lock (lockObject)
{
    if (_currentTask is not null)
        return;

    _currentTask = InitializeTask();
    _currentTask.Start();
}
```

```
if (Volatile.Read(ref _currentTask) is not null)
    return;

lock (lockObject)
{
    if (Volatile.Read(ref _currentTask) is not null)
        return;

    Volatile.Write(ref _currentTask, InitializeTask());
    _currentTask.Start();
}
```

Double-Check Locks (2)

You need to incorporate memory barriers with a Double-Check Lock, otherwise the CPU might reorder the InitializeTask() call and assignment to the field.

This would result in the first check of the log reading a not-fully initialized object.

Resilience and async await

- Polly.Core offers the **ResiliencePipeline** class for resiliency.
- Prefer **ResiliencePipeline.ExecuteAsync** as otherwise, retries will block the caller.
- Cancellation Tokens can be forwarded.
- When you await **ResiliencePipeline.ExecuteAsync**, you will continue once the whole operation including the retries, etc. are finished.

Demo Time!

Outbox Processor

Async void methods

- Sometimes, async void methods are unavoidable, especially when triggering long-running operations.
- Within async void methods, always use a try-catch block because otherwise, you might miss exceptions (because callers are not able to await the task).

What have we learned so far?

- When we want to trigger long-running operations, we must not return the corresponding task.
- We pass it to an async void method and return early.
- When using async void methods, always use try-catch blocks, otherwise you might miss exceptions (the caller cannot await the corresponding task).
- Use a dedicated **CancellationTokenSource** to allow other callers to cancel the long-running operation – don't forget to dispose it.

Epilogue

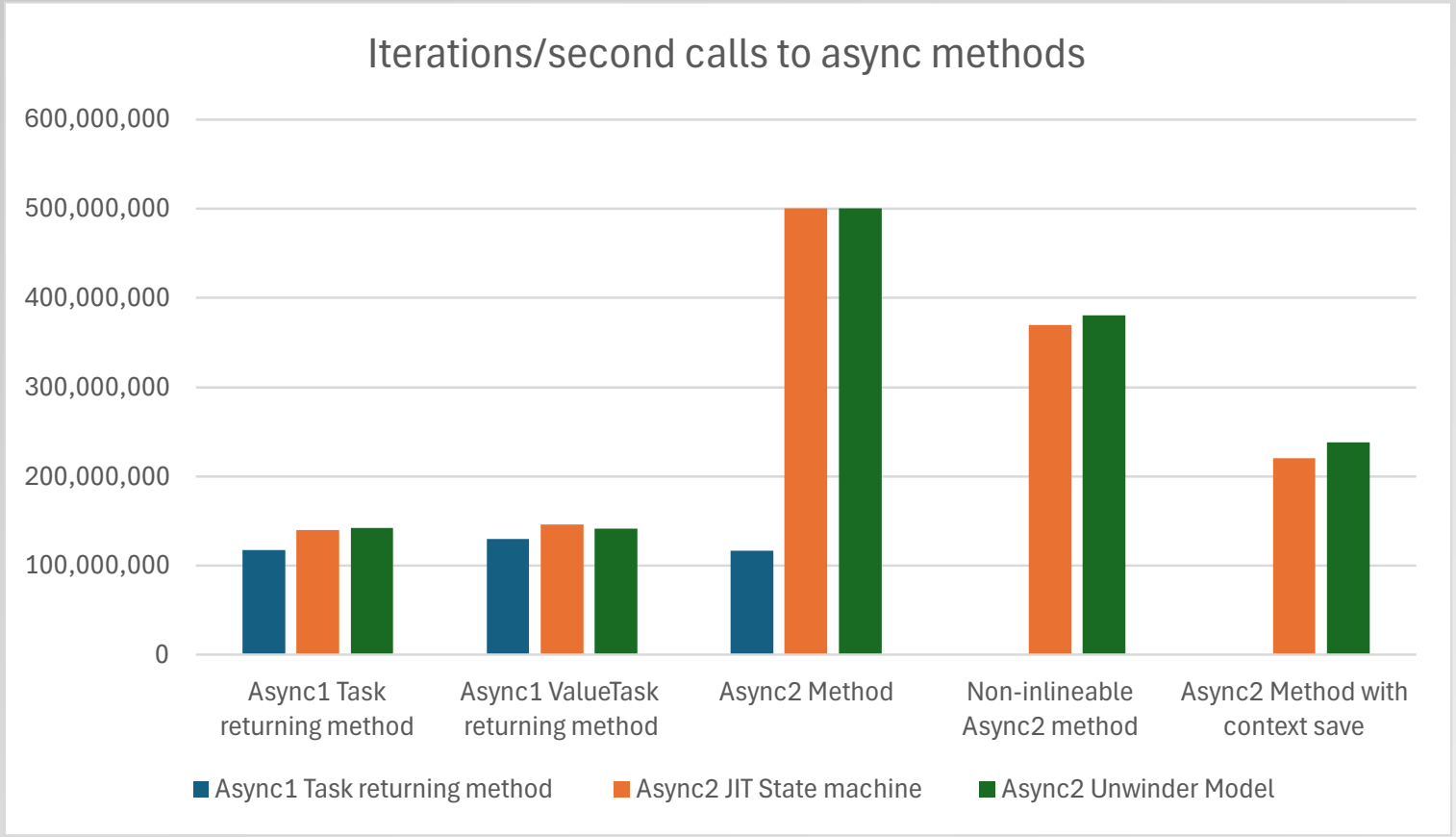
Closing thoughts and future trends

About TaskCompletionSource

- TaskCompletionSource can be used to return an unfinished task to a caller (usually not in an async method).
- Use (Try-)SetResult, (Try-)SetException, and (Try-)Cancel to set a finished state on the Task Completion Source.
- Usually, use TaskCreationOptions.RunContinuationsAsynchronously for your Task Completion Sources – this way continuations will be enqueued on a new thread on the thread pool.
- If you do not use this options, continuations will by default run on the same thread where you call one of the aforementioned methods – this could lead to deadlocks, thread-pool starvation or corruption of state.

Async2 - a new implementation

- In 2023 and 2024, the .NET team conducted research for **Green Threads** and subsequently **Async2**.
- **Green Threads** were inspired by Project Loom and its Fibers in Java. Fibers are not bound to OS-Threads, the scheduling happens directly in the CLR (when the runtime detects that a fiber blocks, it will run a different fiber on the underlying OS thread).
- Green Threads won't be implemented in .NET as there are too many issues with the existing async model.
- However, the state machine will be moved to the CLR resulting in less code generated by the C# compiler.



Threading and asynchronous programming is hard

- Implicit knowledge about the intrinsics of the .NET thread pool, threading, synchronization primitives, and the async await state machine are required.
- Side effects can be hard to track, especially when continuations run on different threads.
- Source code does not indicate this, async await might mislead you as code looks “synchronous”.

We need new principles, especially for young devs

Learn the Internals (LTI)

Study the internals of the runtimes, tools, and frameworks/libraries that you use and understand how they solve recurring problems. Examine which call patterns will result in problems / undesirable outcome and ensure that these patterns are avoided in your code base.

Respect the Process Boundary (RPB)

Make a clear distinction between in-memory and I/O operations as the latter usually have longer execution times and are more error-prone. Consider performing I/O asynchronously so that the calling threads can perform other work while an I/O operation is ongoing. I/O calls should be abstracted so that in-memory logic can run independently from third-party systems.

Sources

- Jeffrey Richter: Advanced Threading in .NET
- Joseph Albahari: Threading in C#
- John Skeet: Asynchronous C# 5.0
- Stephen Toub: How async await really works
- David Fowler: Async Guidance
- Stephen Cleary: There is no thread
- Konrad Kokosa: Pro .NET Memory Management
- Steven Giesel: async2 - The .NET Runtime Async experiment concludes

Thank you very much!
I look forward to your feedback!

https://forms.office.com/e/rHZAXxAmZX

Scan me!

ppedv