.NET async await – In Depth

Thoroughly understand async await to scale your apps

My Coding Zone
07.04.2023

Kenny Pflug

Content

- Why is async await important for scalability?
- What happens during async I/O and async compute?
- How does the C# compiler transform async methods?
- What happens in memory when an async method executes?
- Q&A

think tecture

Kenny Pflug

- Consultant @ Thinktecture AG
- ASP.NET Core Web APIs
- Performance and Memory Management in .NET apps
- Architecture & Design of distributed Systems
- Cloud Native with Azure, Kubernetes, and Terraform

kenny.pflug@thinktecture.com

thinktecture.com

@feO2x

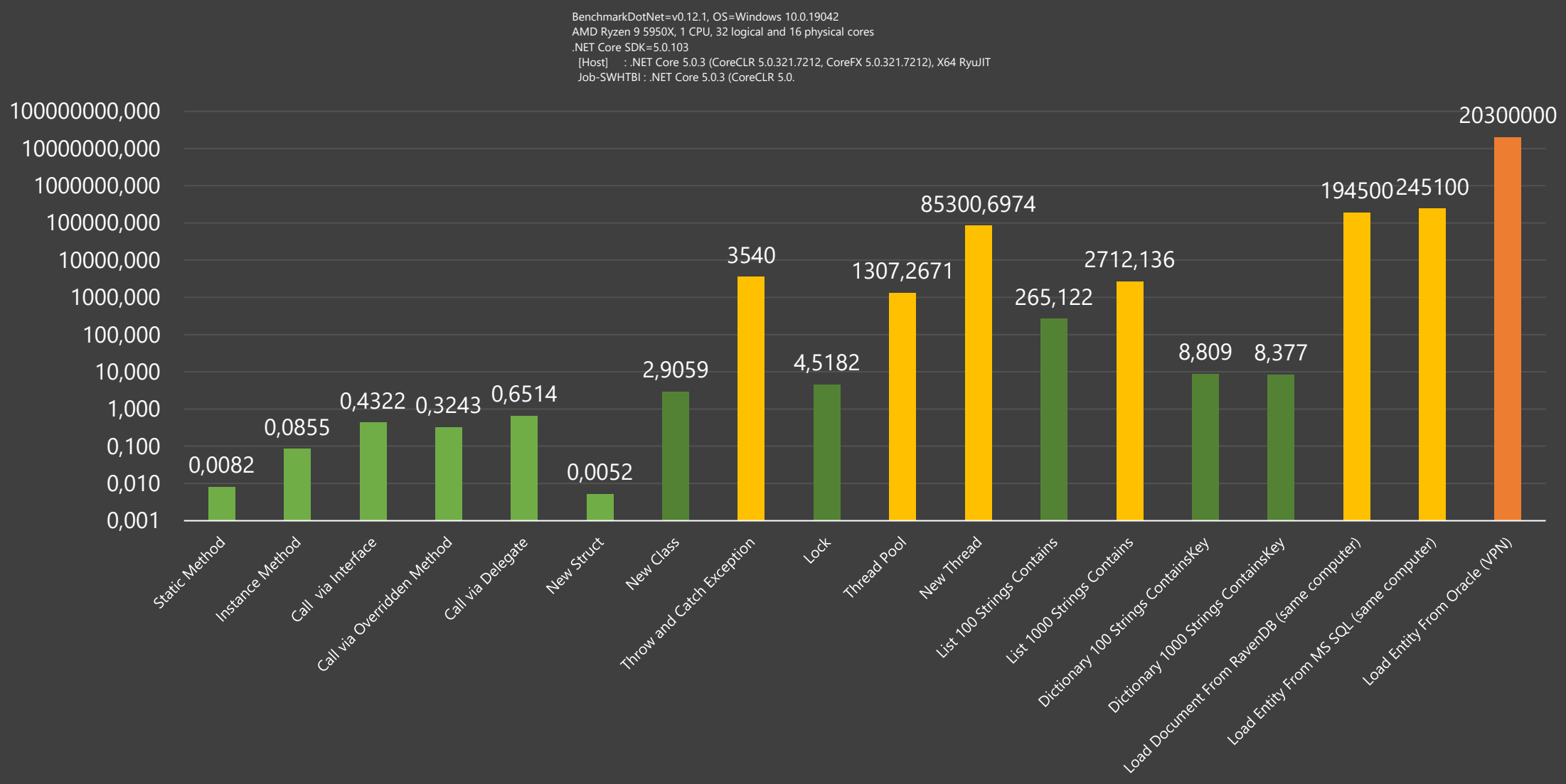
.NET Thread Pool

Why async await is so important for threading and scalability

Live Demo

async await and service scalability

Performance of everyday things



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 (but Green Threads might be coming)
- 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 Miss)
- 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 its worker threads!
It will create more if work items need to be handled by it.

The big issue with the .NET thread pool

What blocks threads?
Thread.Sleep and sync I/O!

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, more will be created
- Depending on the amount of work items and the blockage length, 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 allocates less worker threads and decreases its number earlier towards the CPU Core goal

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

Targeting different third-party systems with async I/O

- HttpClient is async by default
- Socket supports async
- FileStream now supports async 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
- 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: everything supports async I/O except transactions (might be coming to Microsoft.Data.SqlClient)
- Npgsql: full async support
- RavenDB.Client: full async support
- MySql.Data: no support at all, you should use MySqlConnector instead
- Oracle.ManagedDataAccess: no async support, but might be coming with the 22 release, there are paid third-party access libraries (I haven't tried them)
- SQLite: no async support (but only file-based)
- SAP Connector for .NET: no async support

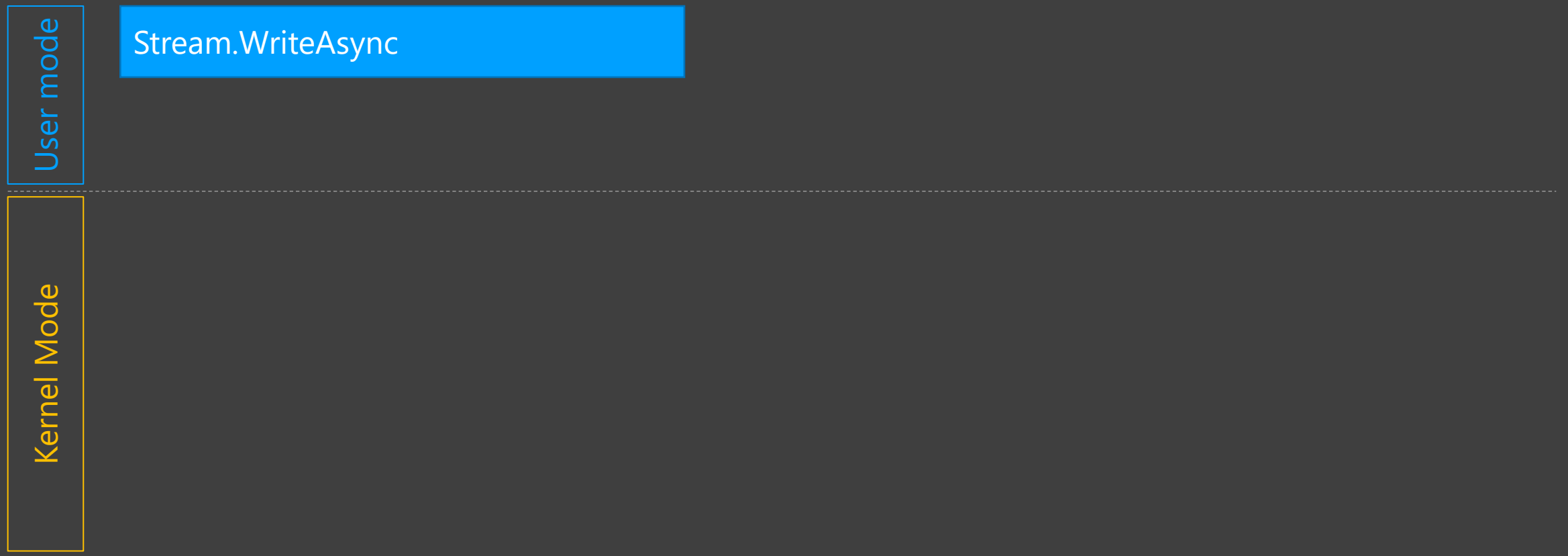
Async I/O in detail

Two groups of async operations

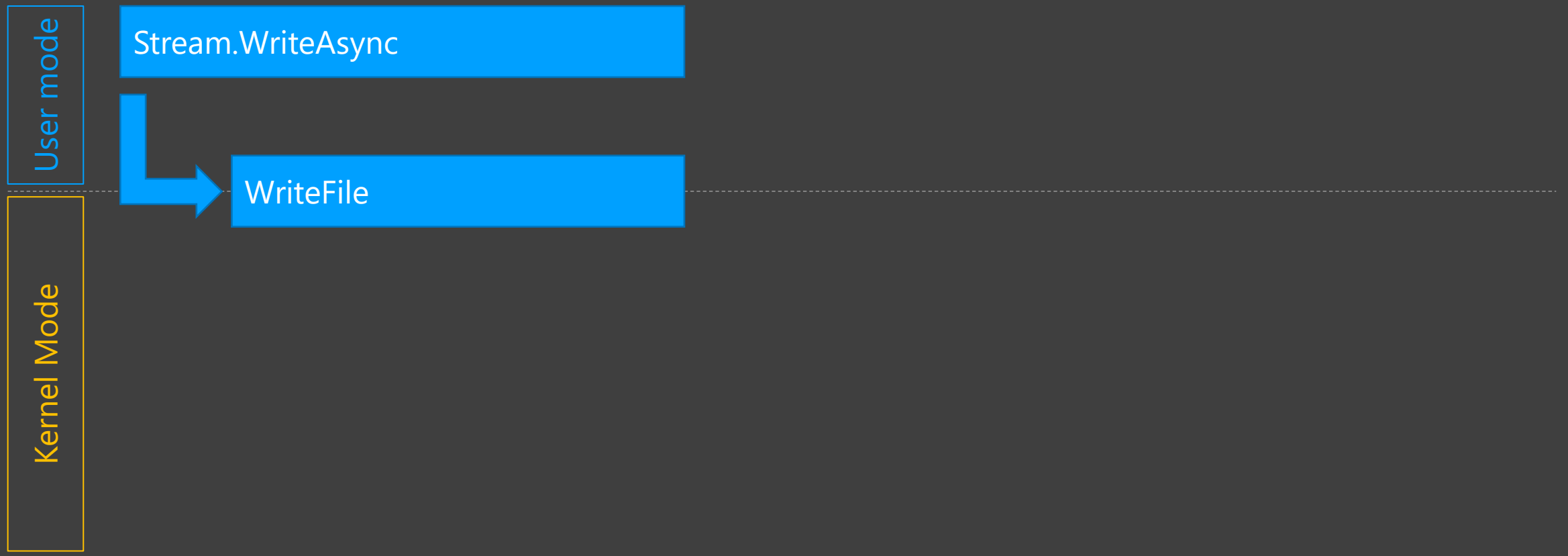
Async I/O

Async Compute

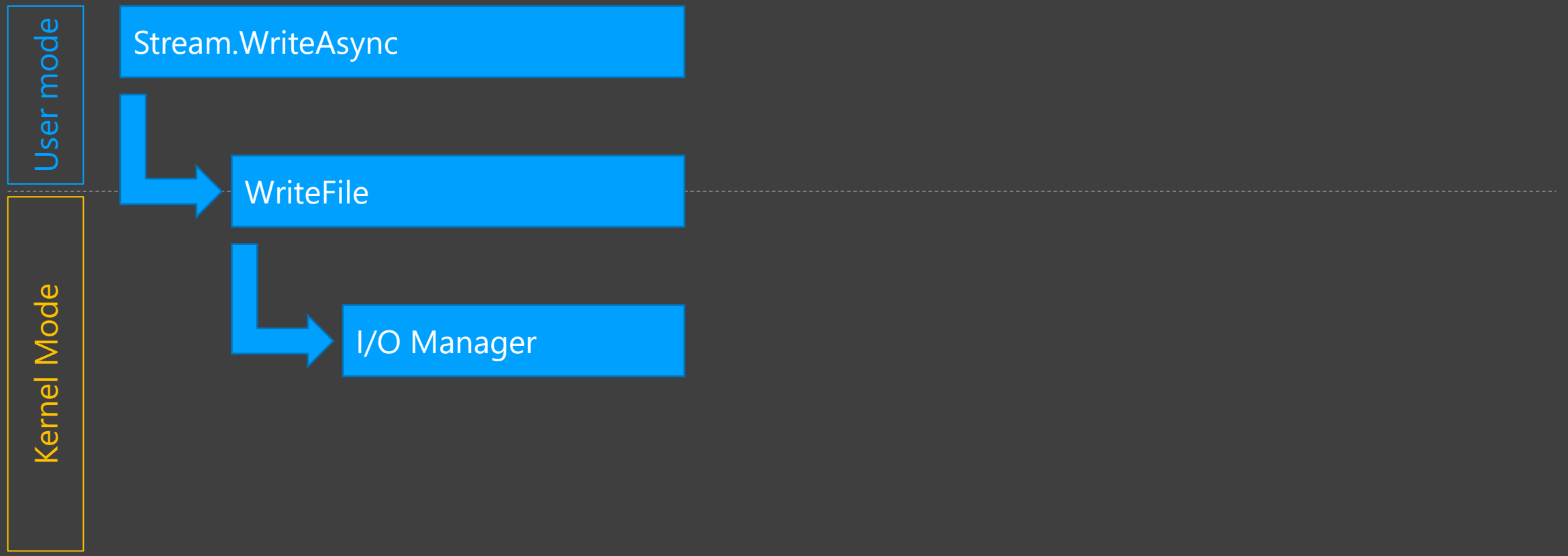
Async I/O Write File operation on Windows



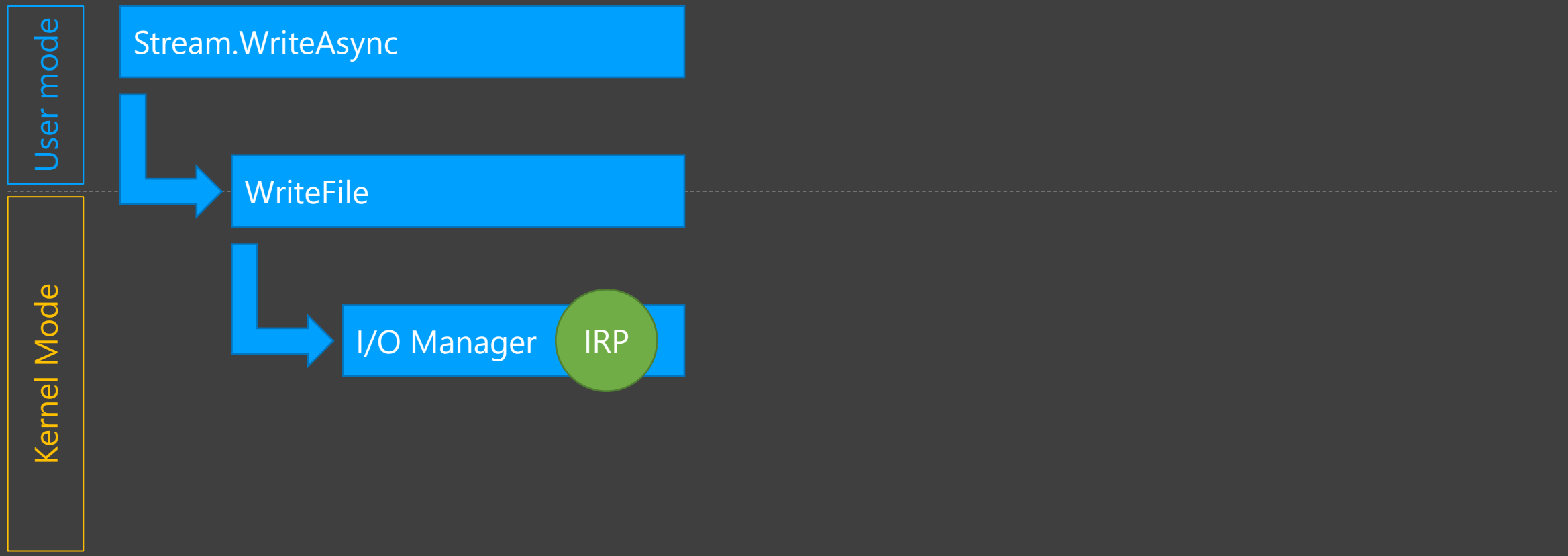
Async I/O Write File operation on Windows



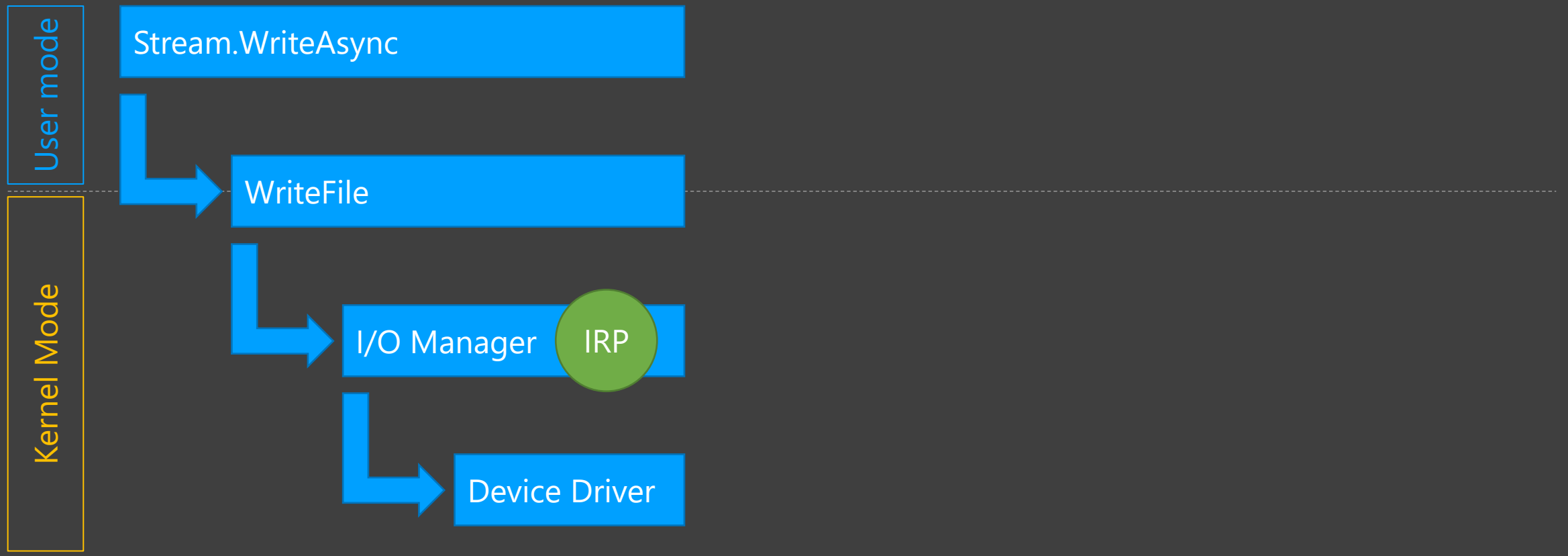
Async I/O Write File operation on Windows



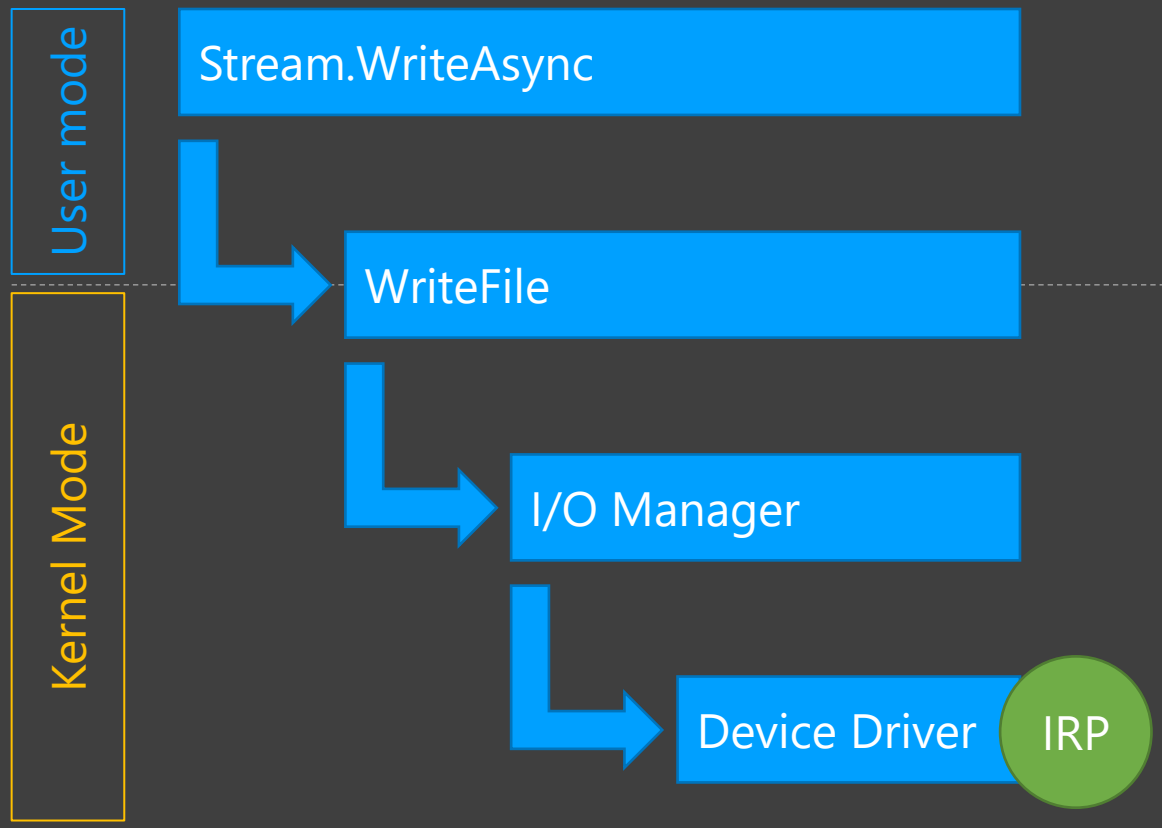
Async I/O Write File operation on Windows



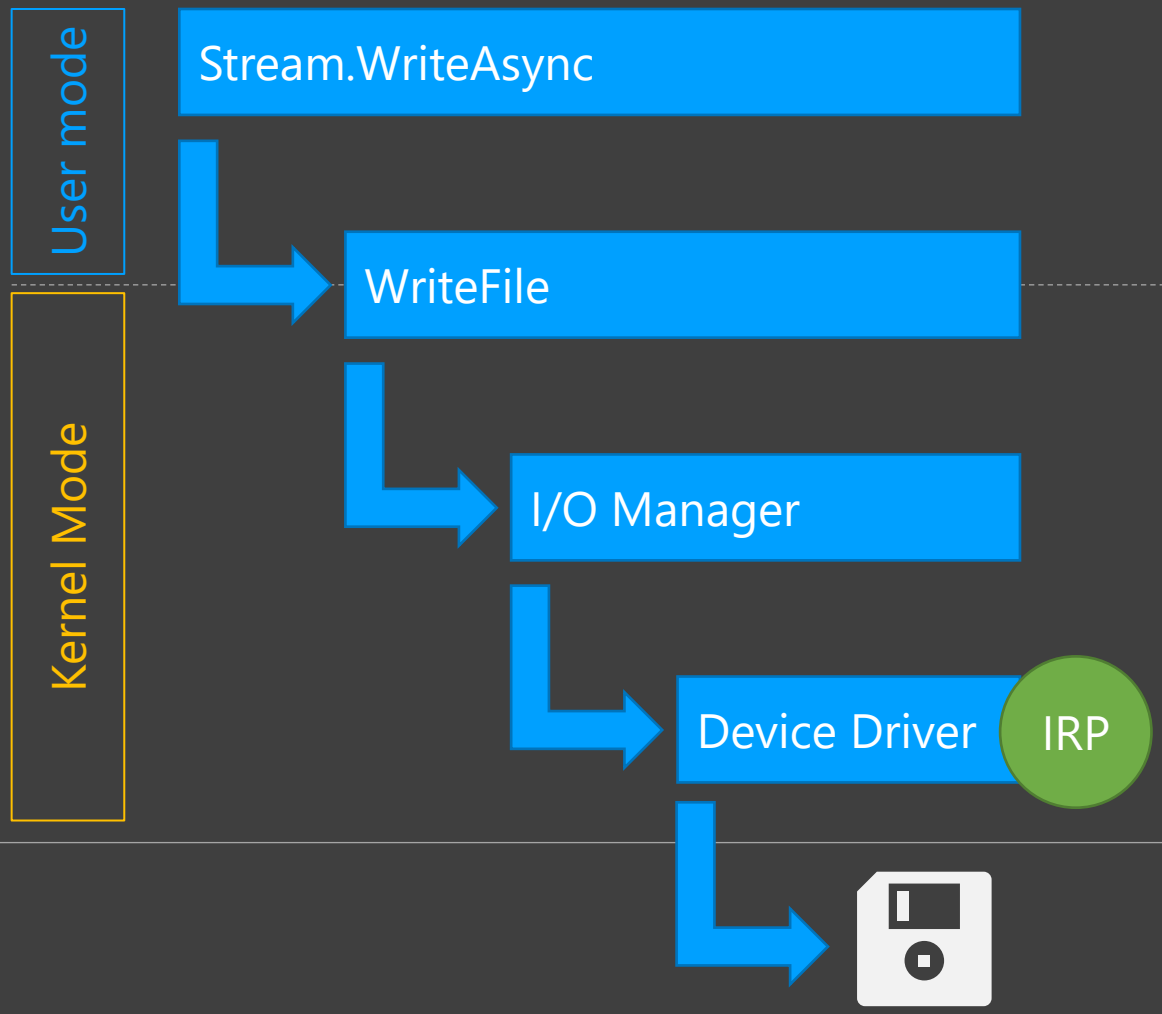
Async I/O Write File operation on Windows



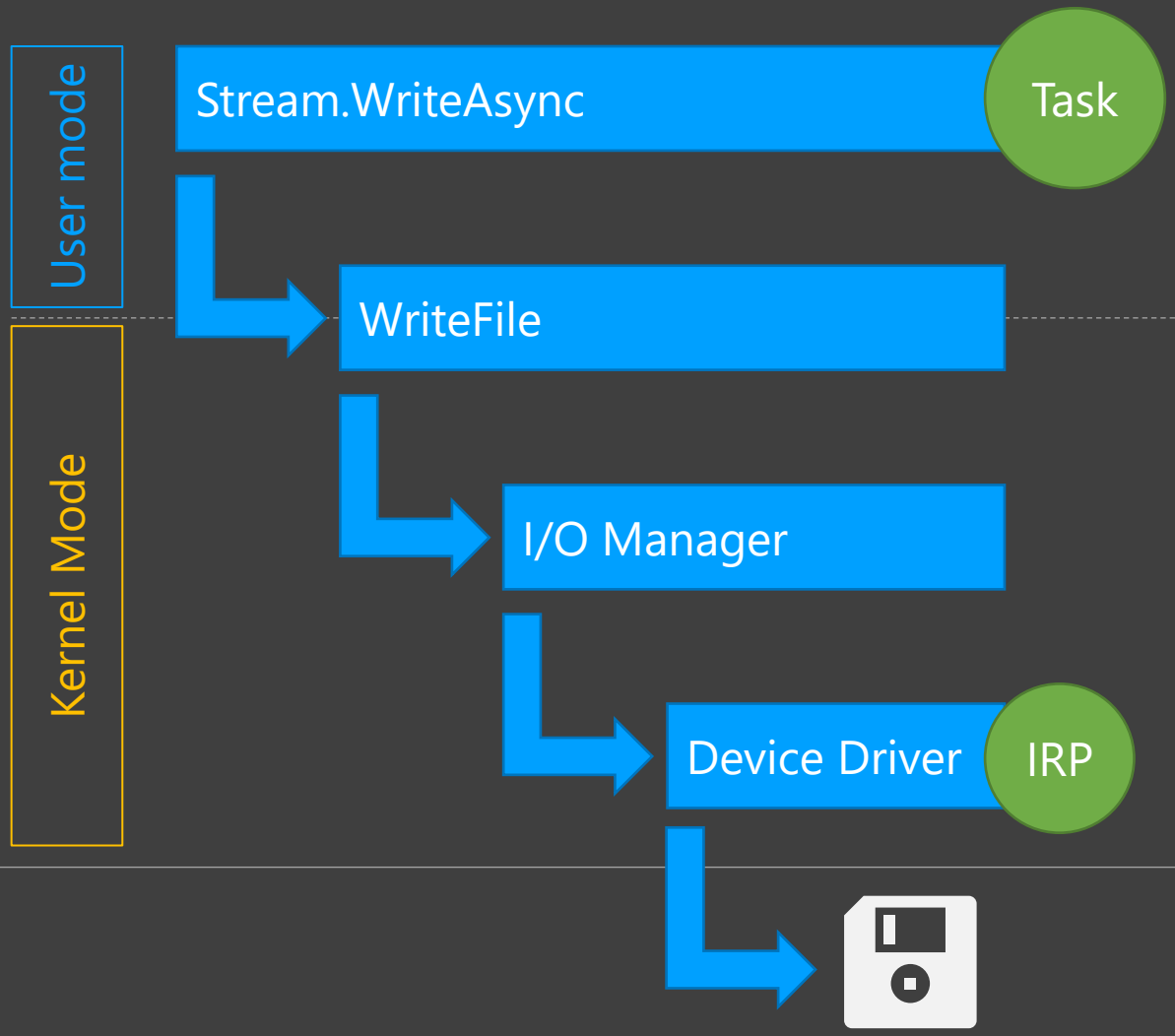
Async I/O Write File operation on Windows



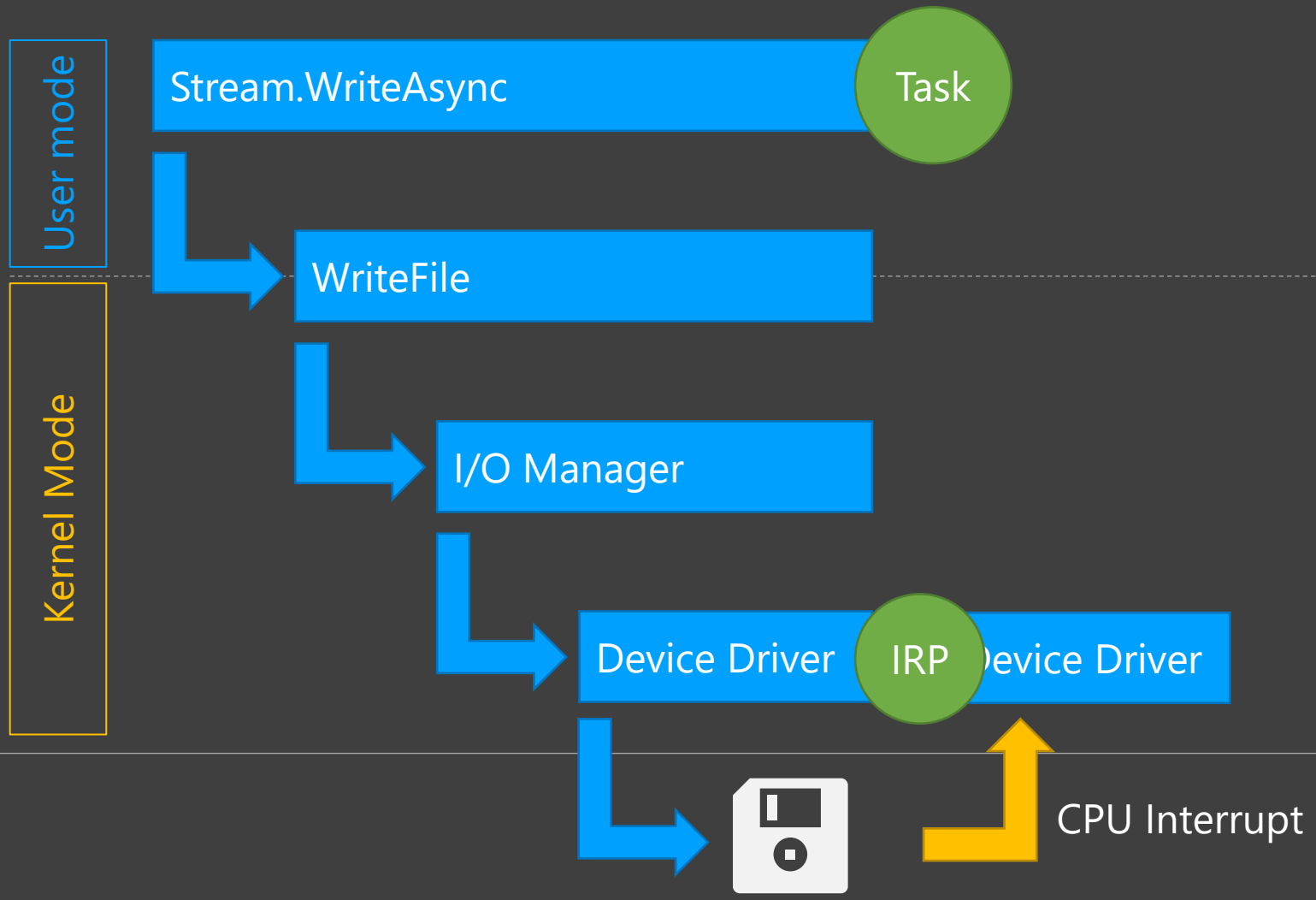
Async I/O Write File operation on Windows



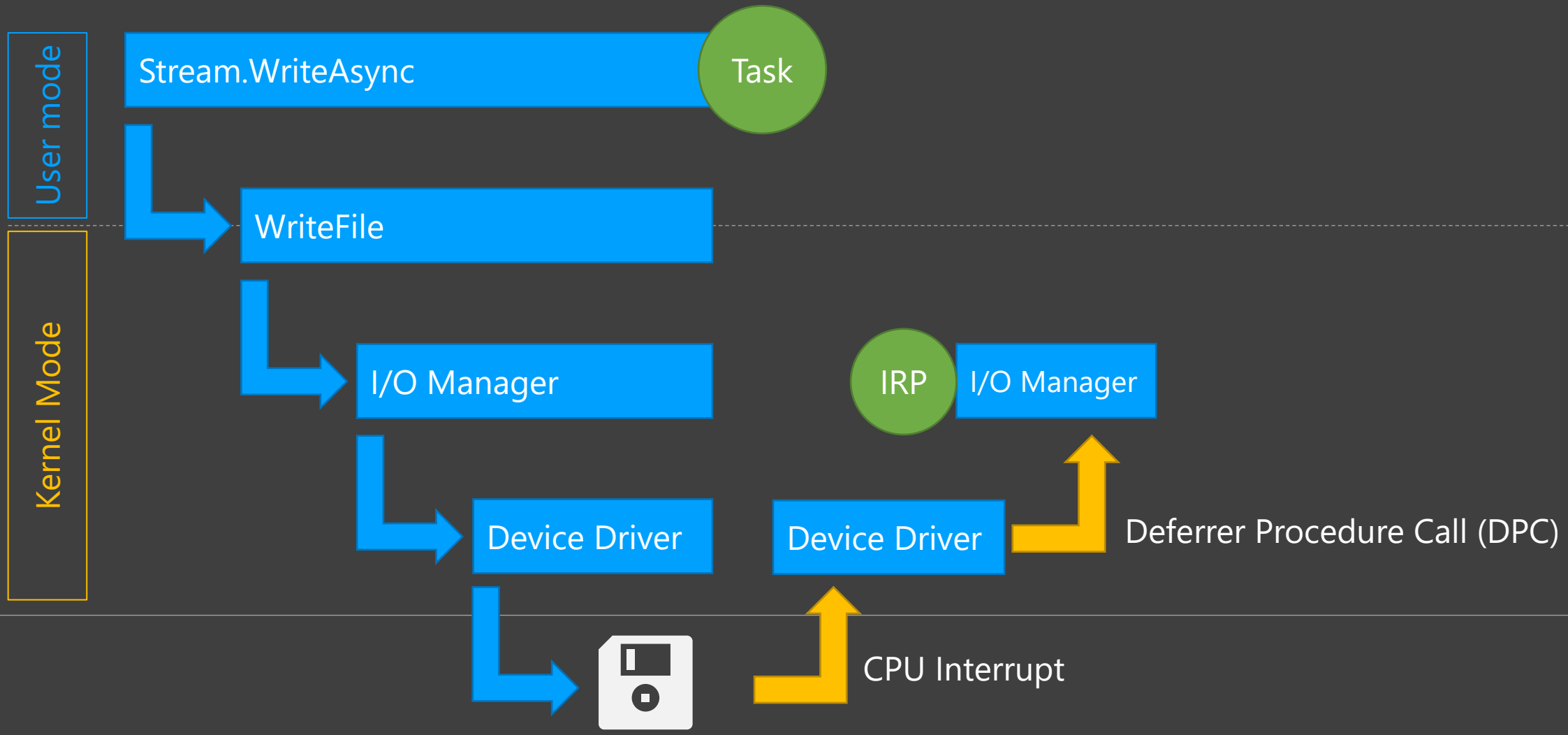
Async I/O Write File operation on Windows



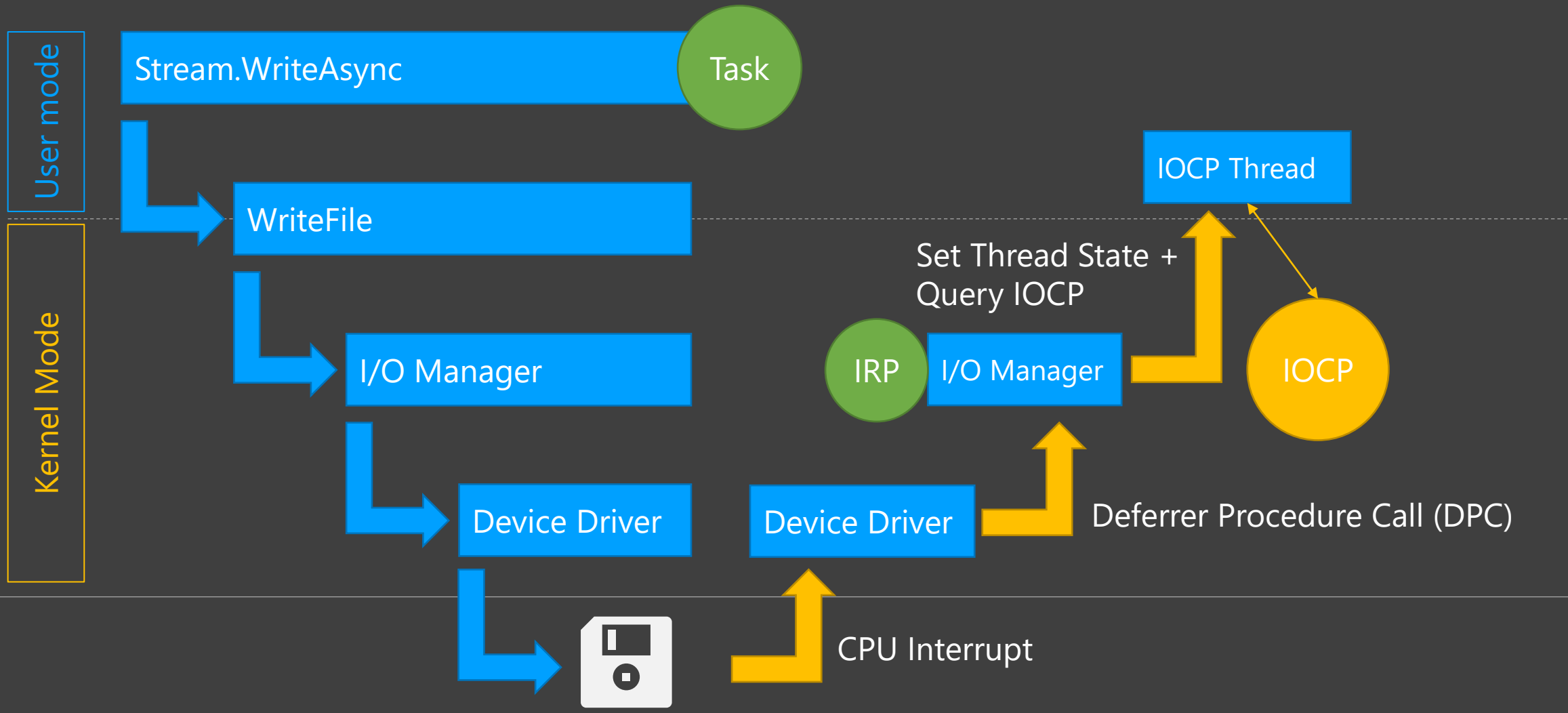
Async I/O Write File operation on Windows



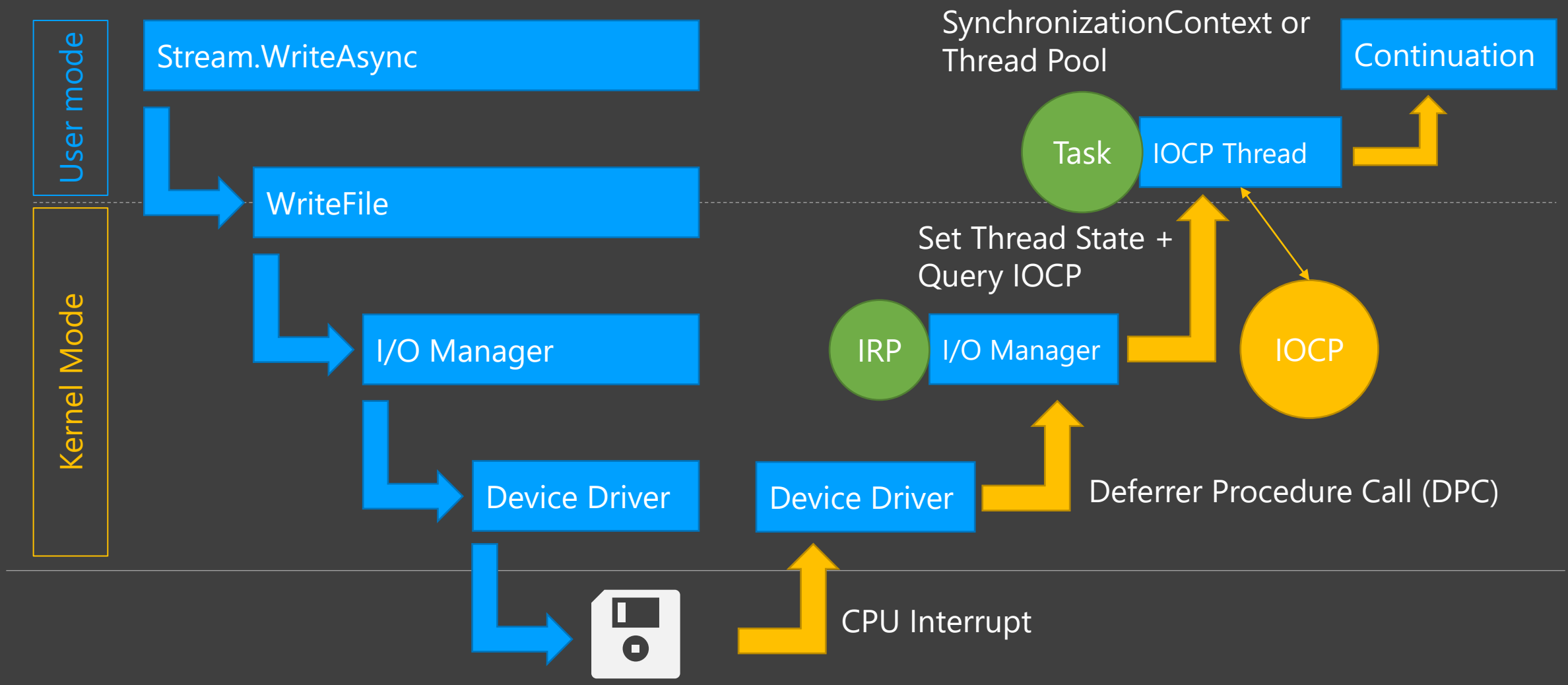
Async I/O Write File operation on Windows



Async I/O Write File operation on Windows



Async I/O Write File operation on Windows



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

Synchronous I/O

- Instead of returning, the caller will be blocked by the I/O Manager
- Meanwhile, the driver stack will (most likely) perform async I/O (there are no sync drivers?)

async await decompiled

Live Demo

async await decompiled

async Methods

- The C# compiler will transform any method that is marked with the async modifier
- This state machine has a MoveNext method that consists of your actual code and generated code for await expressions (depends heavily on Control Flow)
- The fields of this struct consist of your original method's parameters, variables, the "this" reference, task awaiters, and a method builder (the reusable part of the state machine)
- In Debug mode, the state machine is a class instead of a struct
- It will return to the caller when an async operation is started but not completed
- You can analyze your own methods by using an IL Viewer

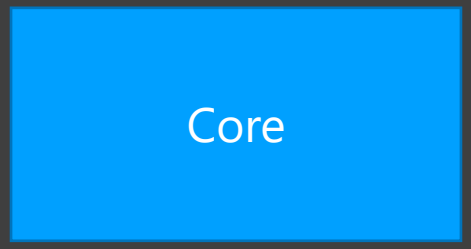
Benefits of async await

- It looks like synchronous code, but executes asynchronously
- No more need for continuation methods, logic can stay in one function
- Ties in with the existing Task Parallel Library (TPL)
- Debuggers let you step over await statements as if they synchronous

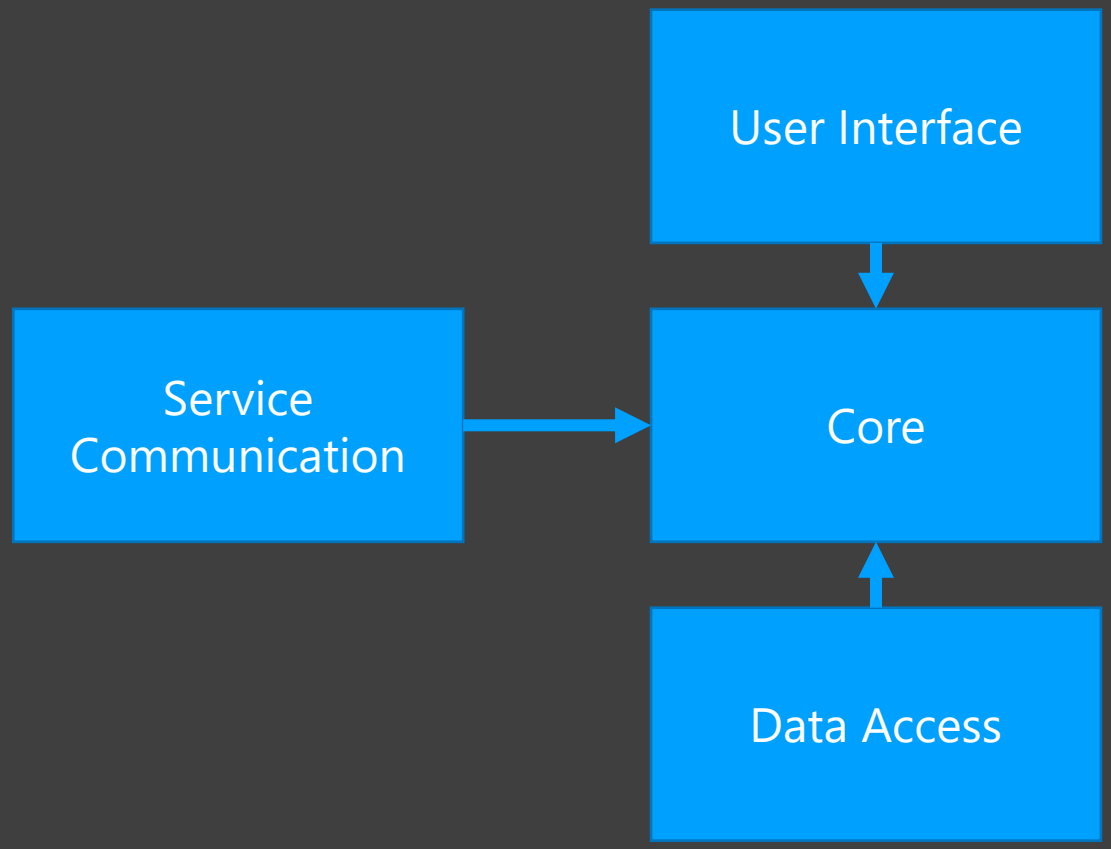
async await – what you need to be aware of

- The control flow is drastically altered:
  methods can return when an await expression executes
- 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)
- async await has a drastic impact on our Object-Oriented design

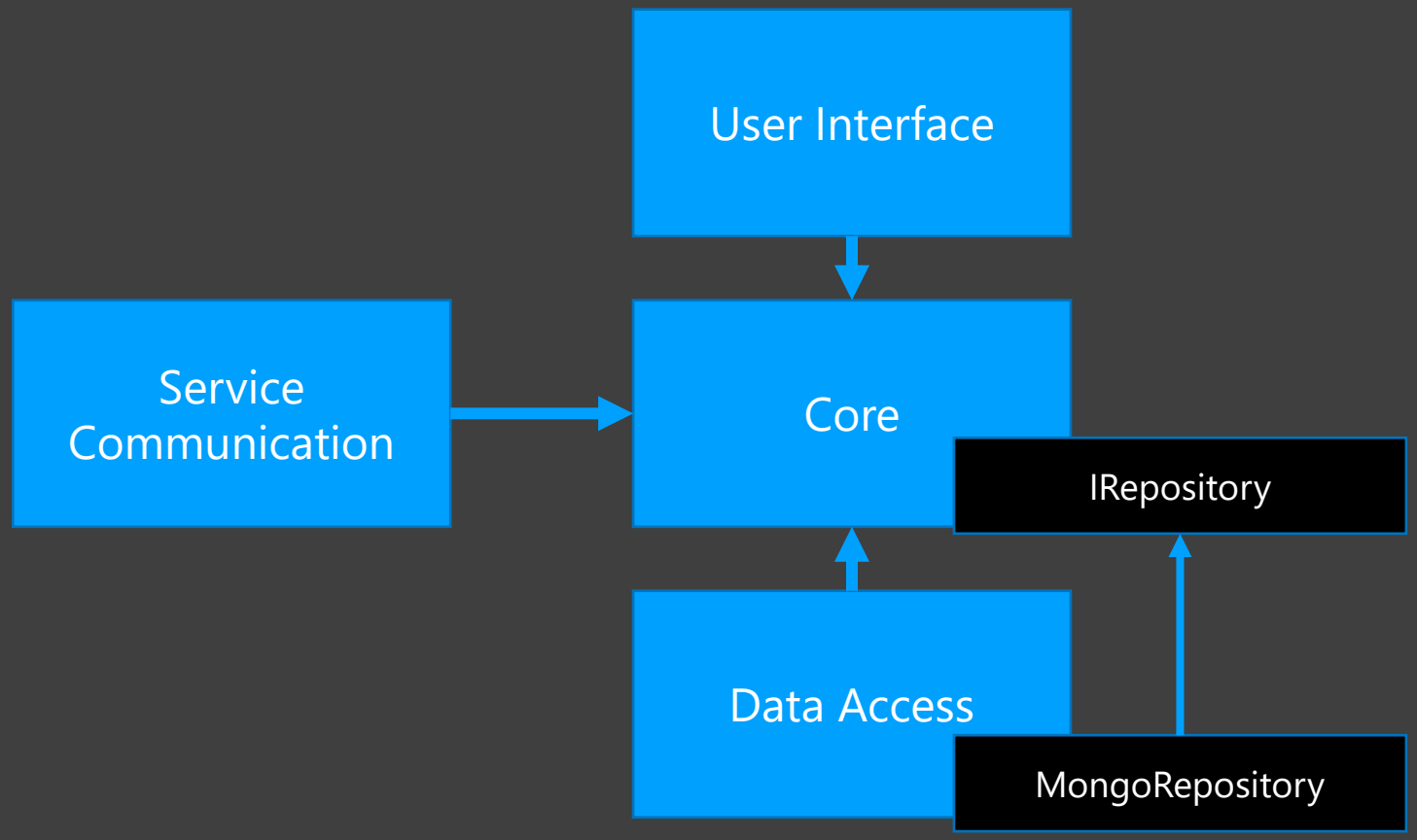
async await and O-O Design



async await and O-O Design



async await and O-O Design



async await and O-O Design

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

public class MongoRepository : IRepository
{
    public Contact LoadById(Guid id) { }
}
```

async await and O-O Design

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

```
public class MongoRepository : IRepository
{
    public async Task<Contact> LoadByIdAsync(Guid id)
    { }
}
```

async await and O-O Design

```
public interface IRepository
{
    Task<Contact> LoadByIdAsync(Guid id);
}
```

```
public class MongoRepository : IRepository
{
    public async Task<Contact> LoadByIdAsync(Guid id)
    { }
}
```

A little bit about the Task Parallel Library (TPL)

- The TPL implements Task, Task<T>, ValueTask, and ValueTask<T>, amongst other types
- A method is considered asynchronous when it returns one of the mentioned types (not when it is marked with async)
- If you have an async method that often returns synchronously, consider returning ValueTask<T> to reduce allocations
- You can implement an asynchronous method synchronously by leaving out the async keyword and simply returning one of the following:
  - Task.CompletedTask
  - Task.FromResult<T>(returnValue)
  - new ValueTask<T>(returnValue)
  - default(ValueTask)

async await memory snapshots

What happens in memory when an async method executes?

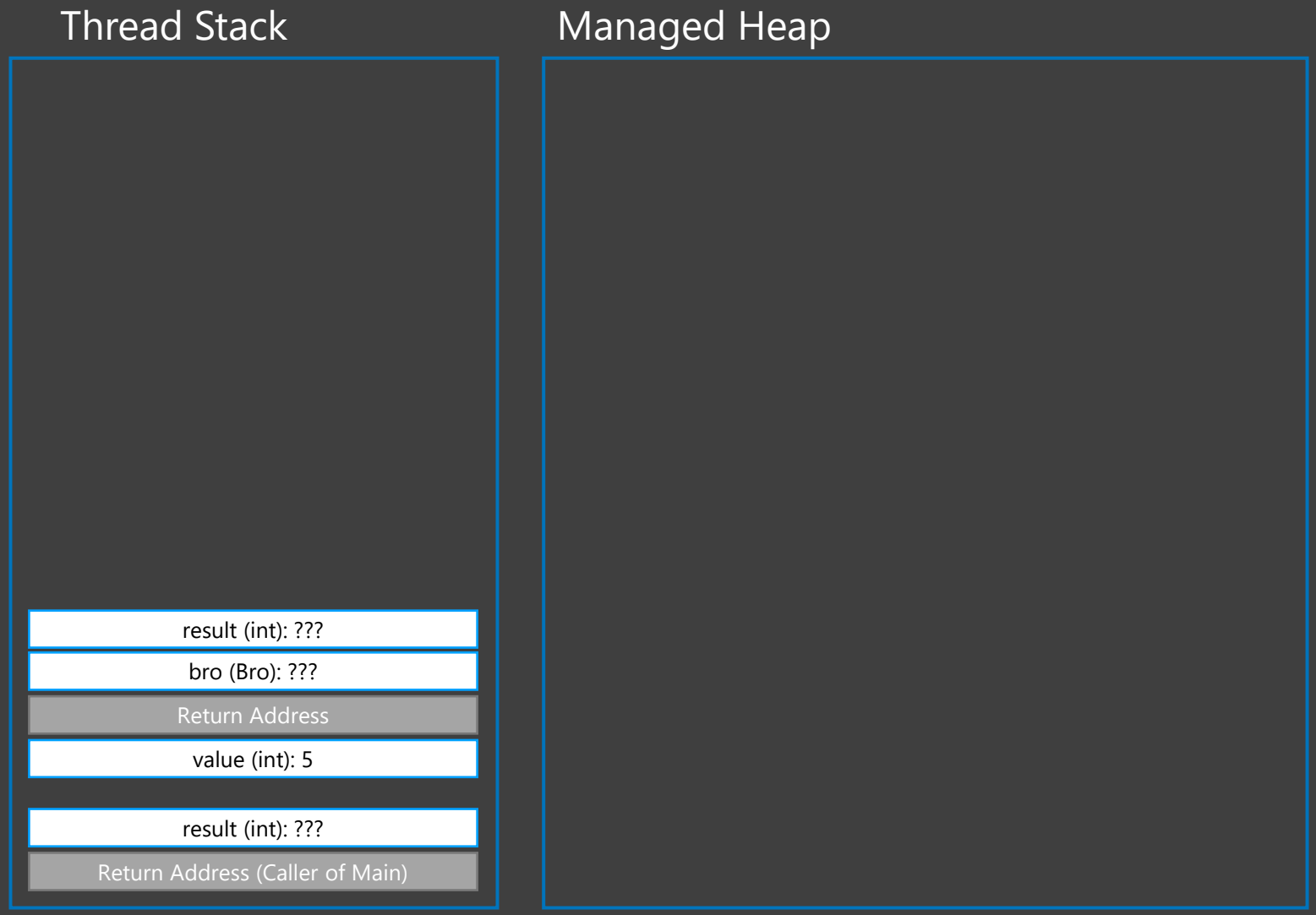
A simple example

Thread Stack

Managed Heap

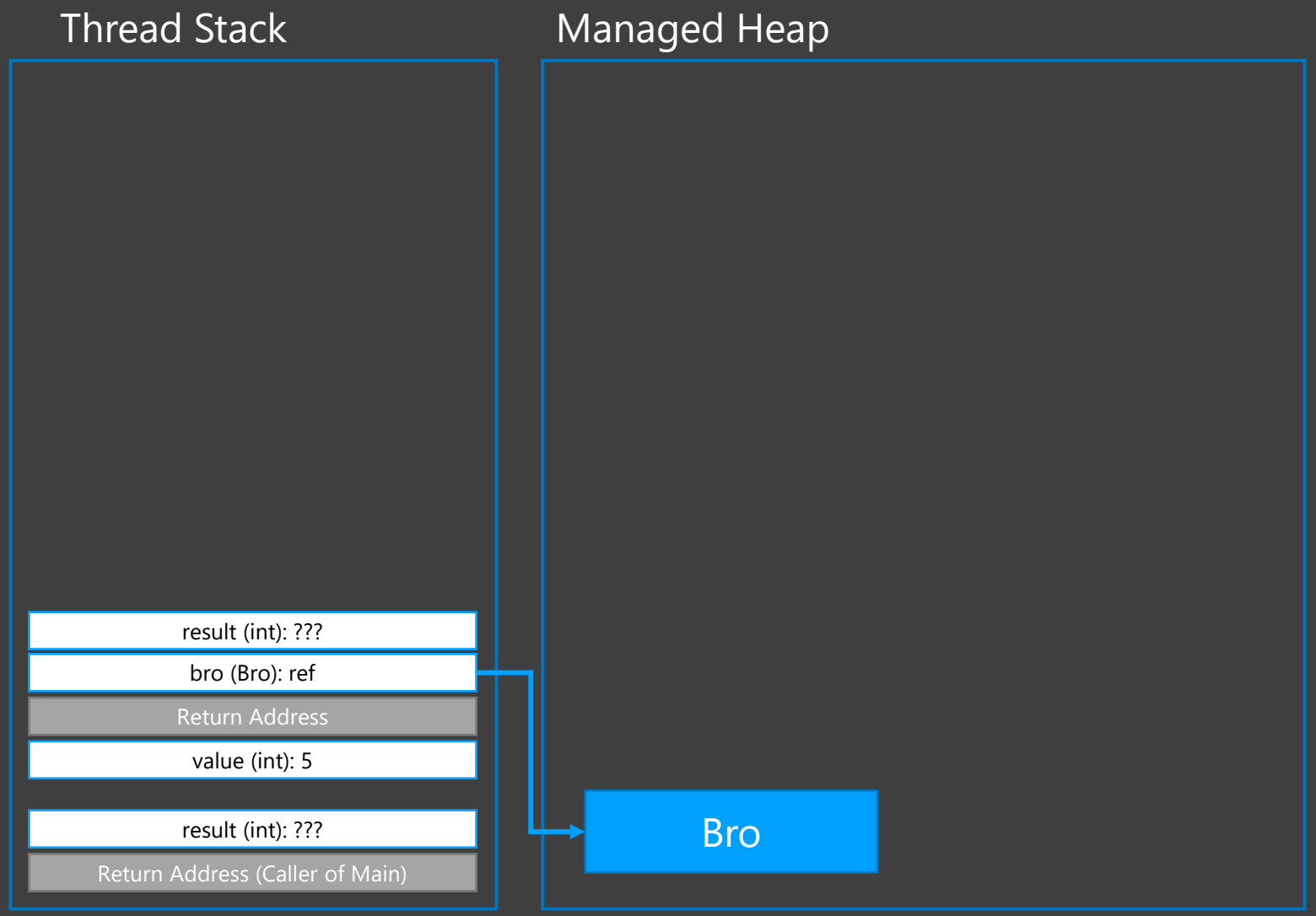

```
public static void Main()
{
    var result = HighFive(5);
}

private static int HighFive(int value)
{
    var bro = new Bro();
    var result = bro.HighFive(value);
    return result;
}

public class Bro
{
    public int HighFive(int value) =>
        value + 5;
}
```

A simple example



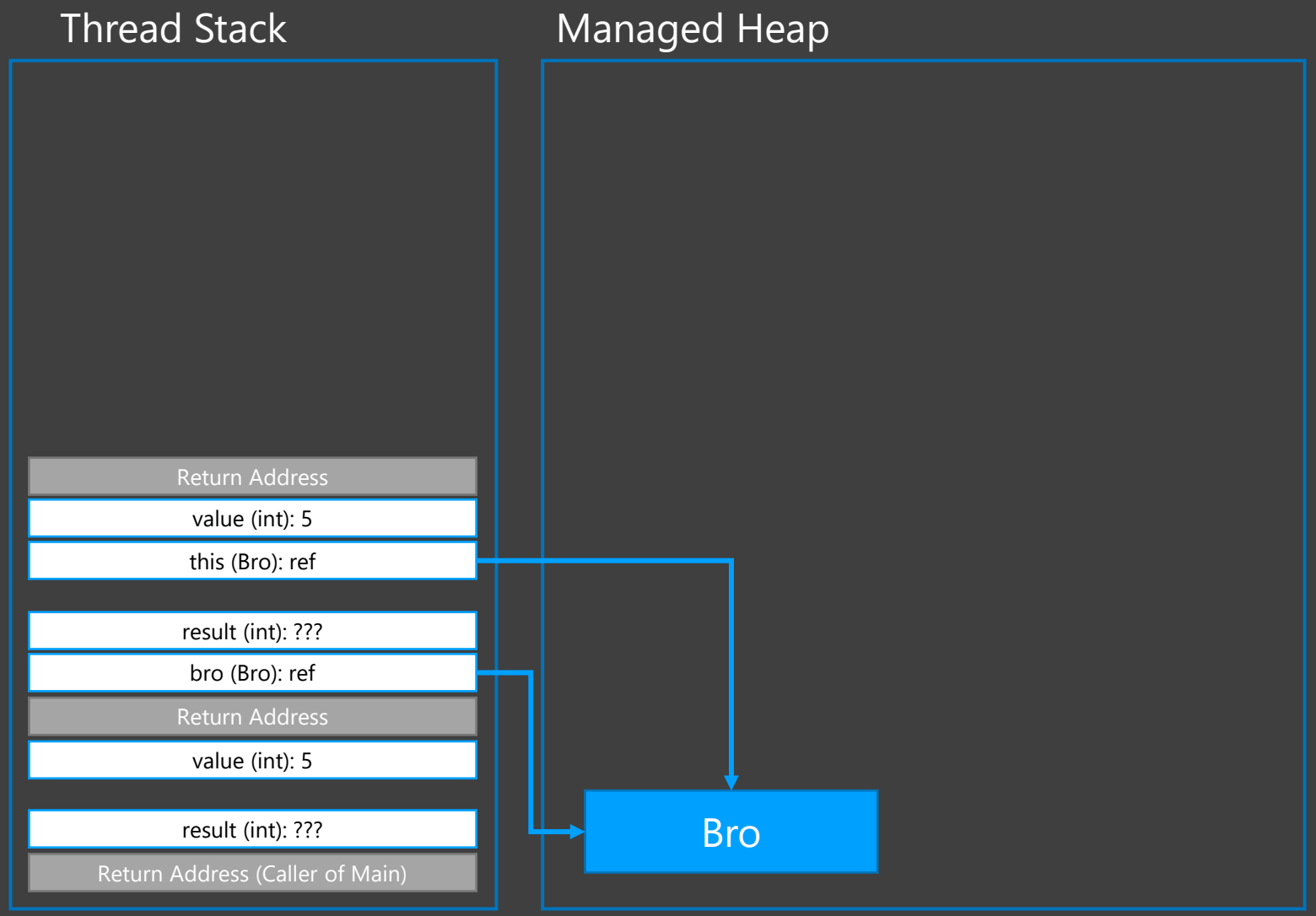

```
public static void Main()
{
    var result = HighFive(5);
}

private static int HighFive(int value)
{
    var bro = new Bro();
    var result = bro.HighFive(value);
    return result;
}

public class Bro
{
    public int HighFive(int value) =>
        value + 5;
}
```

A simple example



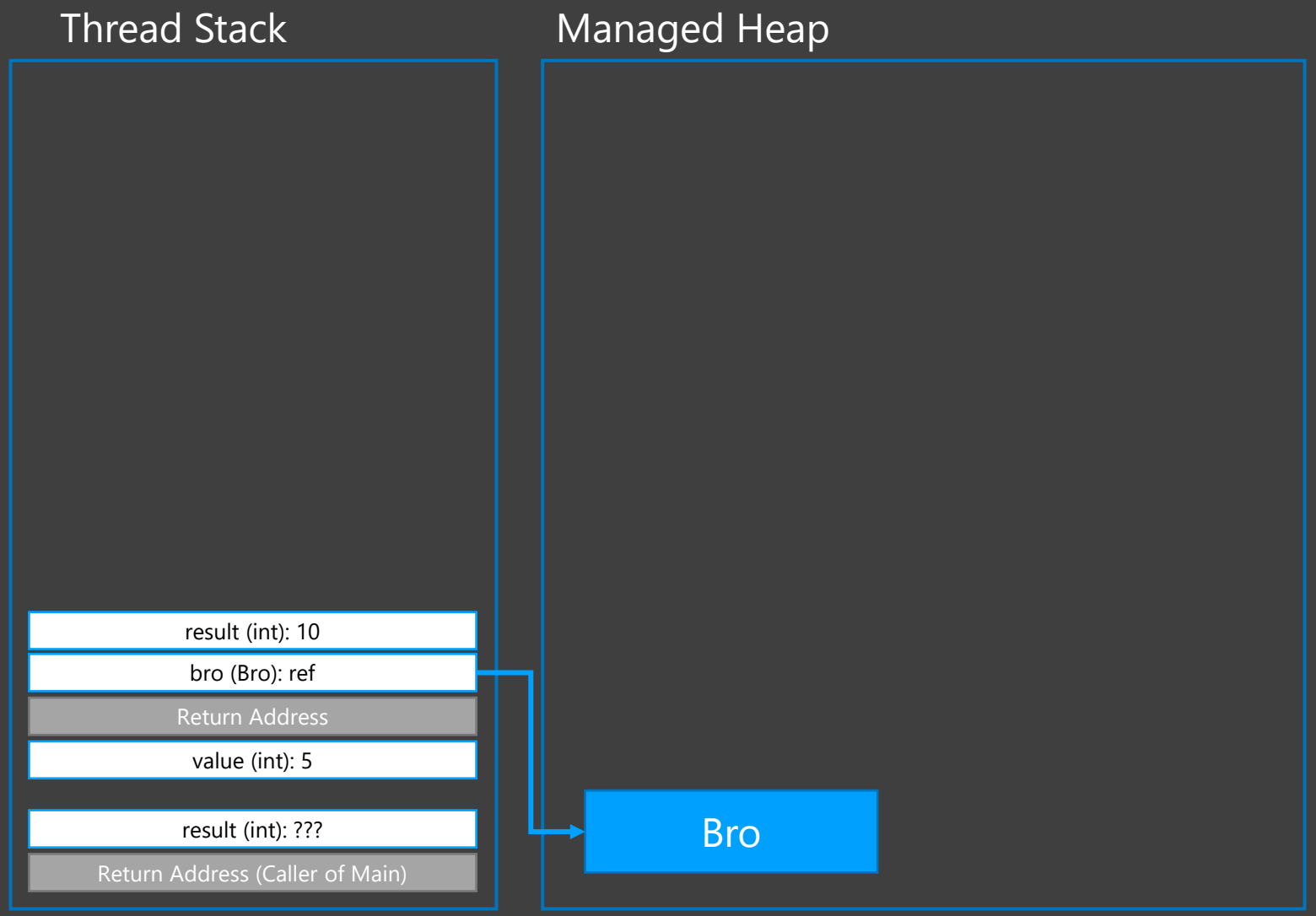

```
public static void Main()
{
    var result = HighFive(5);
}

private static int HighFive(int value)
{
    var bro = new Bro();
    var result = bro.HighFive(value);
    return result;
}

public class Bro
{
    public int HighFive(int value) =>
        value + 5;
}
```

A simple example



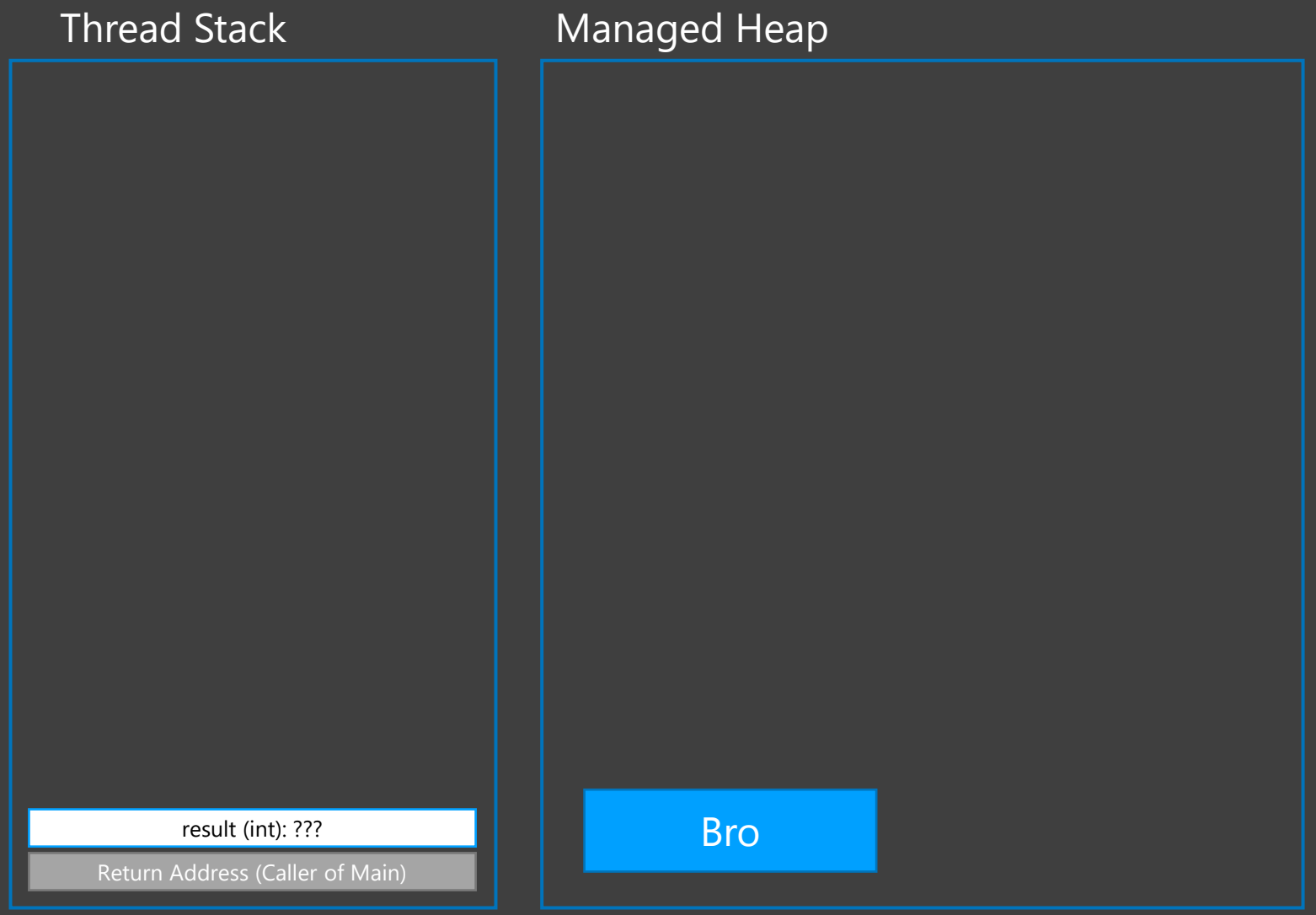

```
public static void Main()
{
    var result = HighFive(5);
}

private static int HighFive(int value)
{
    var bro = new Bro();
    var result = bro.HighFive(value);
    return result;
}

public class Bro
{
    public int HighFive(int value) =>
        value + 5;
}
```

A simple example




```
public static void Main()
{
    var result = HighFive(5);
}

private static int HighFive(int value)
{
    var bro = new Bro();
    var result = bro.HighFive(value);
    return result;
}

public class Bro
{
    public int HighFive(int value) =>
        value + 5;
}
```

A simple example

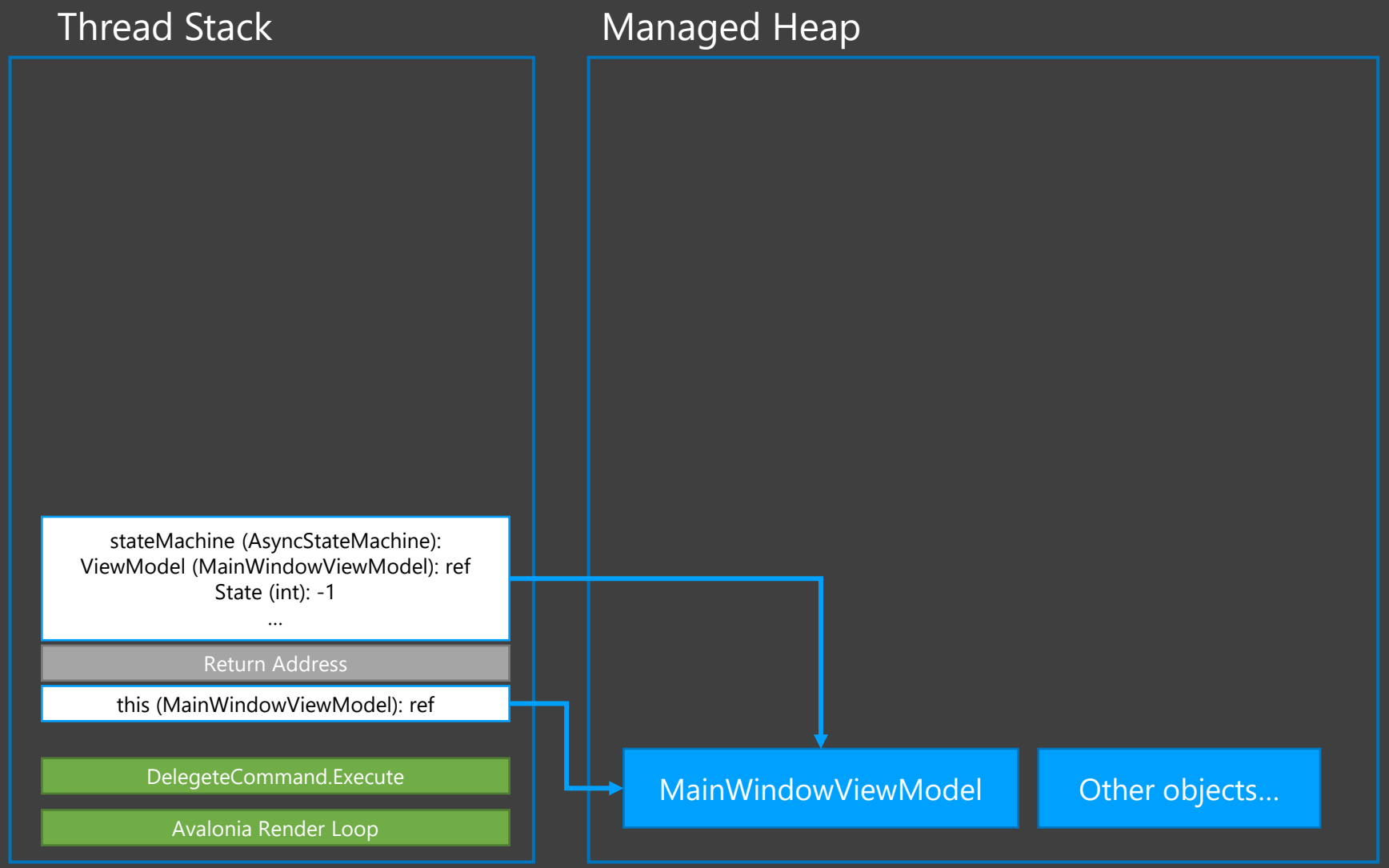



```
public static void Main()
{
    var result = HighFive(5);
}

private static int HighFive(int value)
{
    var bro = new Bro();
    var result = bro.HighFive(value);
    return result;
}

public class Bro
{
    public int HighFive(int value) =>
        value + 5;
}
```

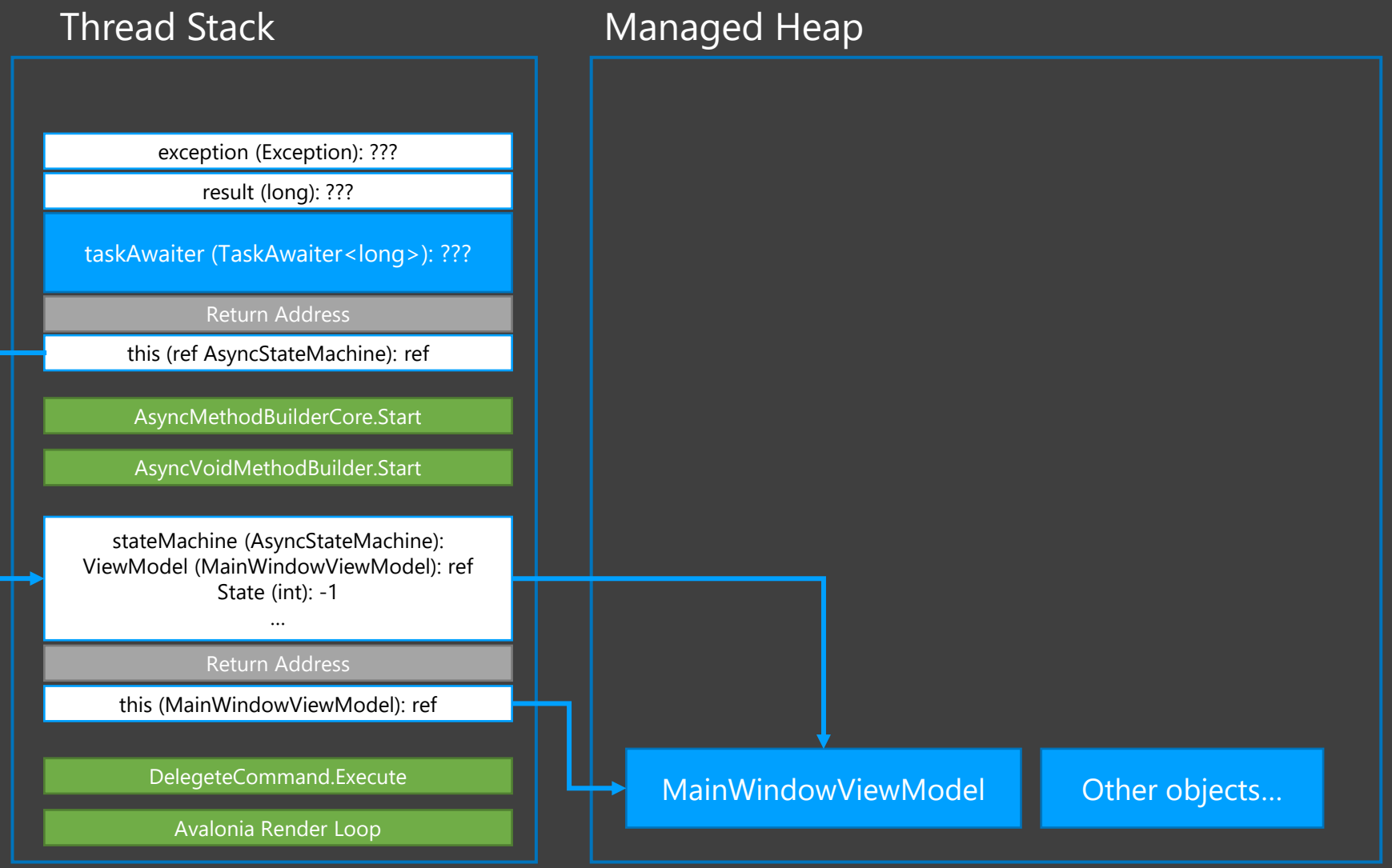
General stuff about memory management

- Each thread has its own stack where parameters and variables for methods are held
- When a new method is called, a so called Activation Frame (or Stack Frame) is pushed
- The Activation Frame consists of
  - all parameters (including the this reference for instance methods)
  - the return address to the caller
  - all variables that are used in this method
- Afterwards, all statements of the method will be executed – once its finished, the Activation Frame is deallocated automatically
- Reference Types (classes, delegates, interfaces) are always instantiated on the Managed Heap (objects)
- Value Types (structs, enums) are either part of an object (field), in a static field, or in a parameter or variable
- If you cast an instance of a Value Type to a Reference Type, then the instance will be boxed on the managed heap
- Unboxing is the reverse process: this happens when a previously boxed instance is cast to its Value Type

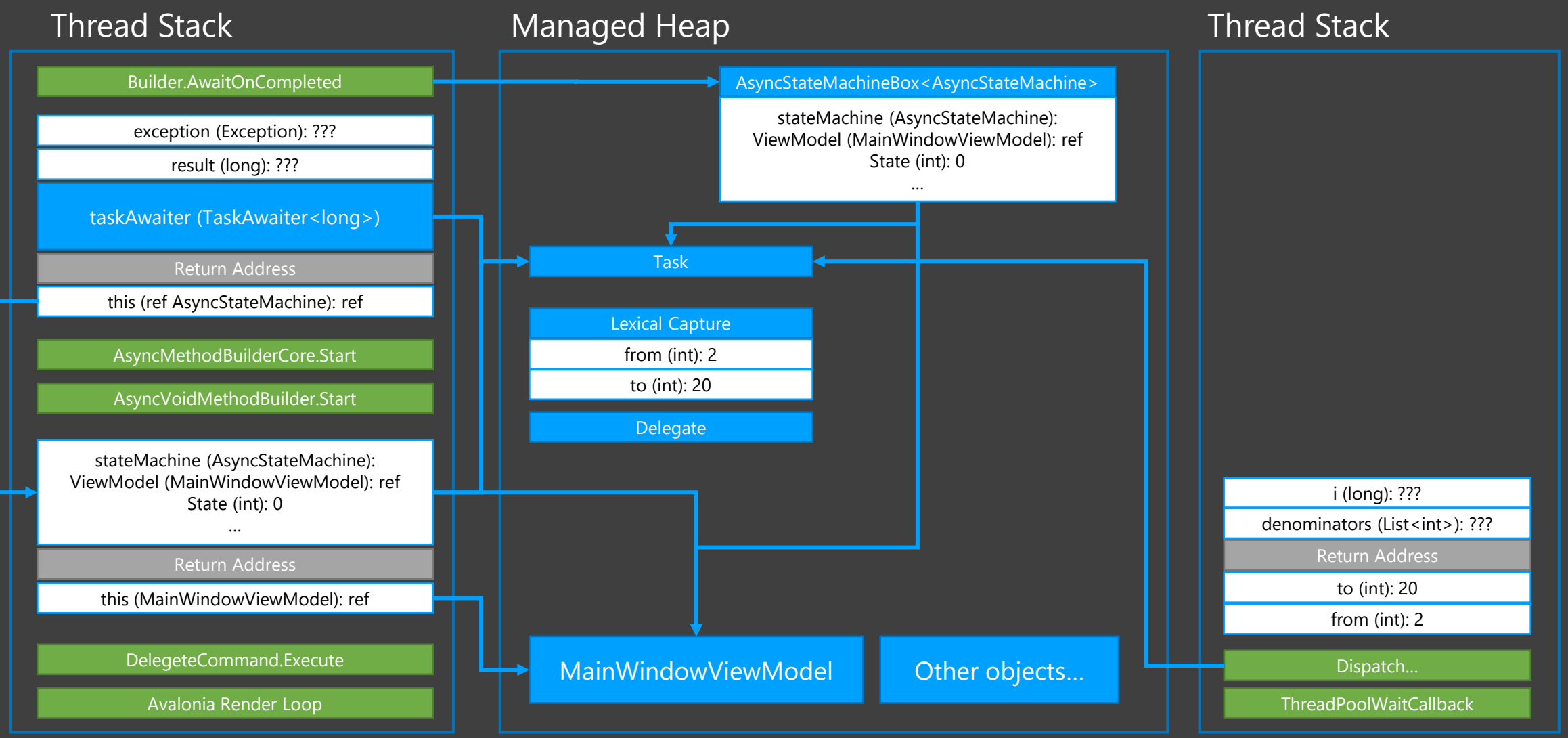
Memory Snapshots: Before calling stateMachine.Builder.Start



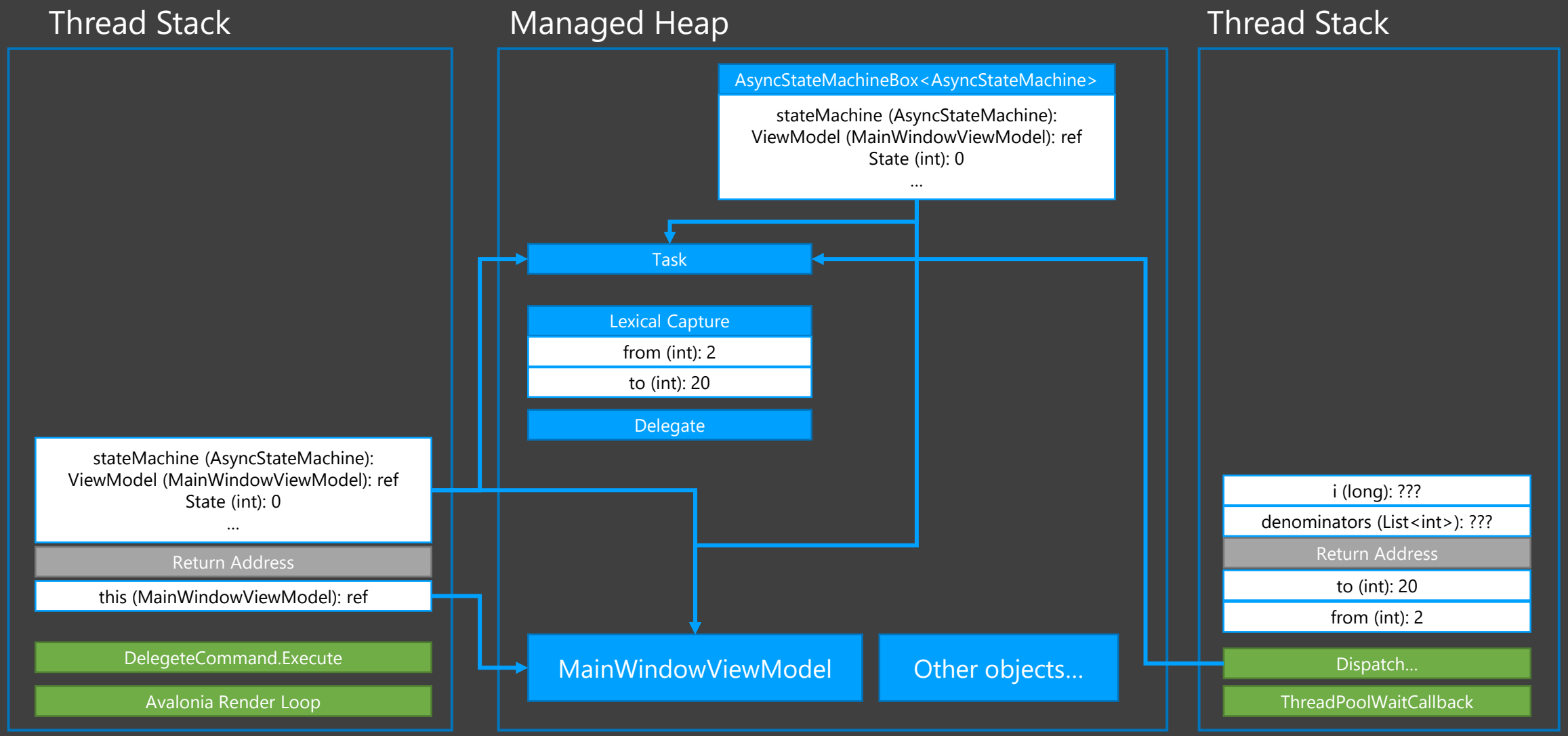
Memory Snapshots: At the beginning of MoveNext



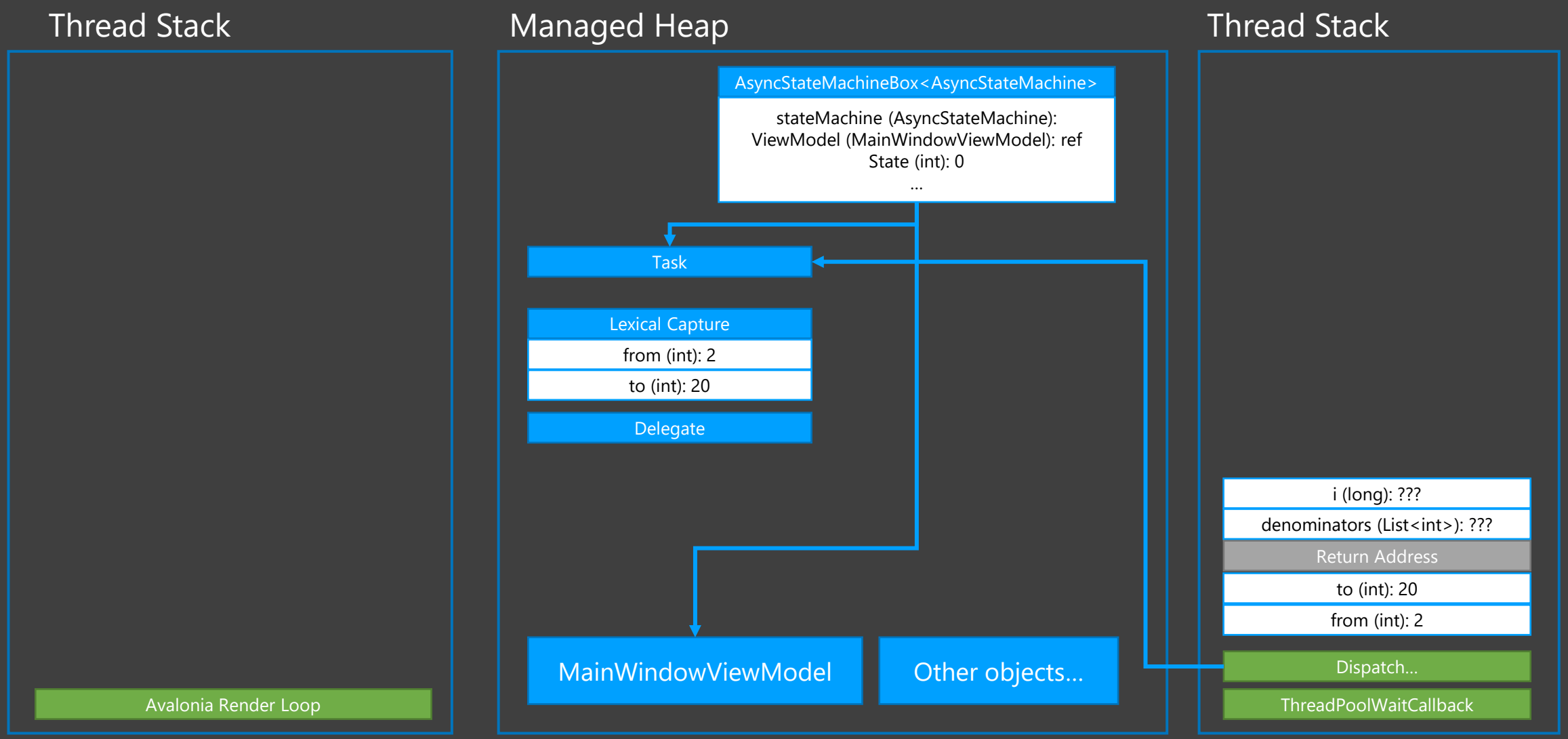
Memory Snapshots: at the end of Builder.AwaitOnCompleted



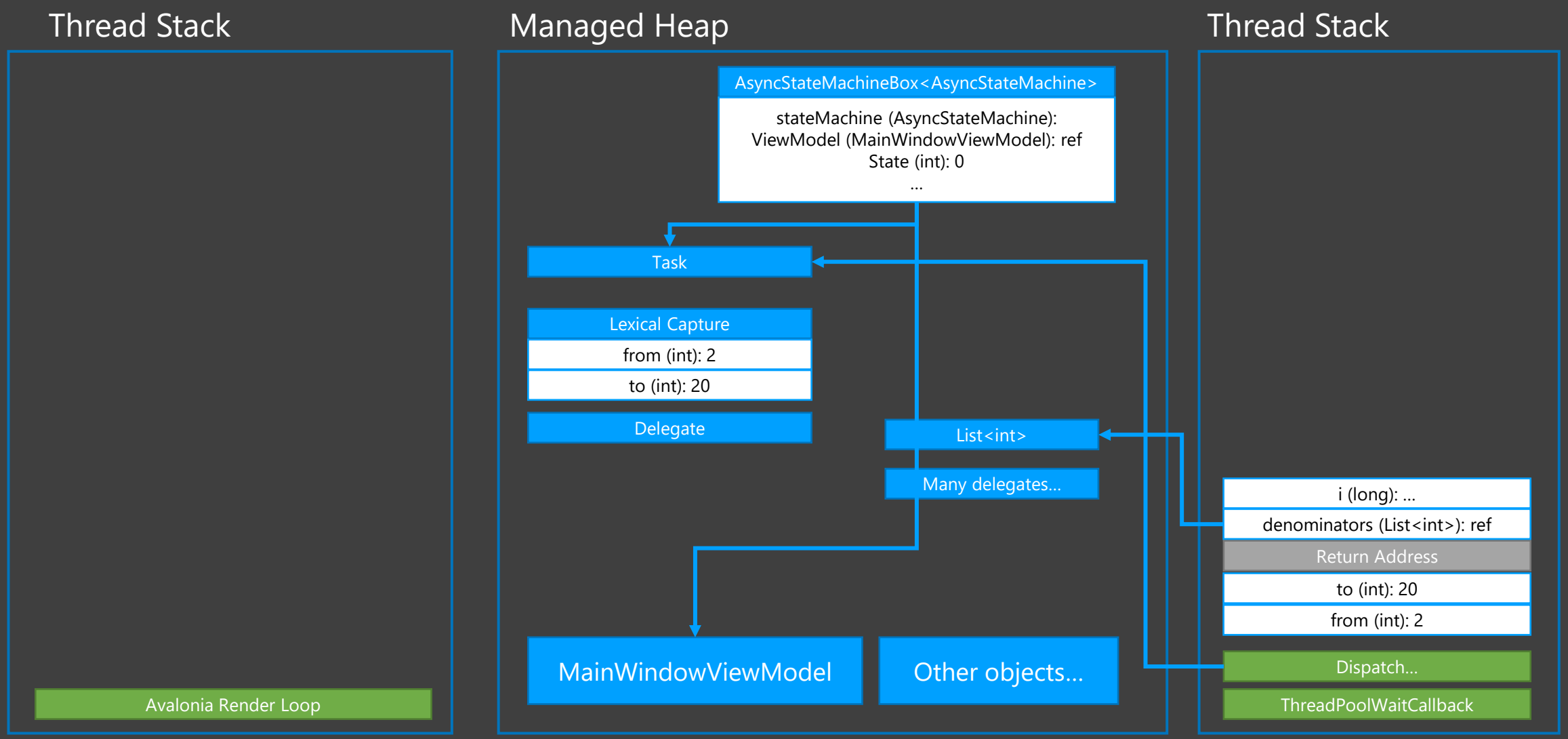
Memory Snapshots: return to calling method



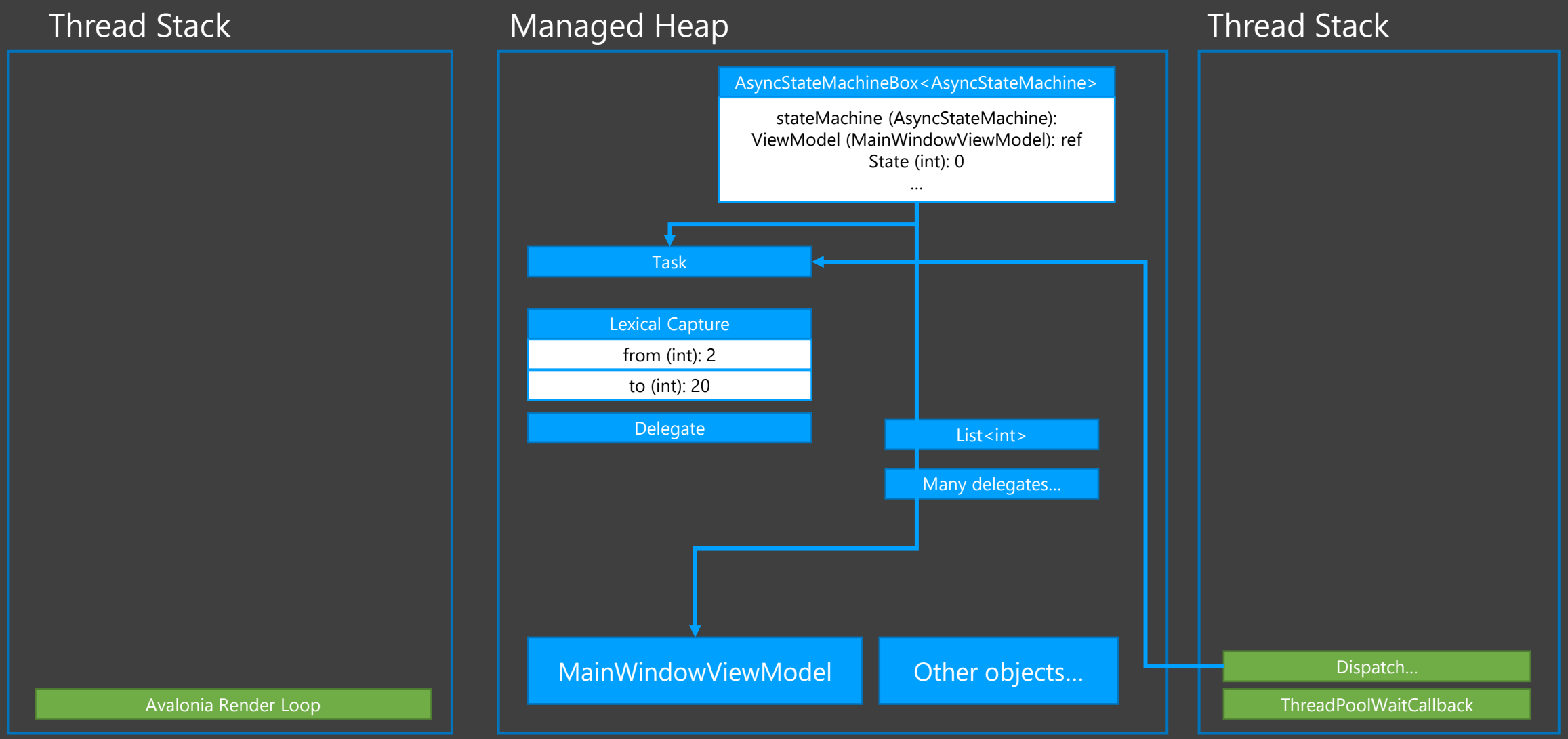
Memory Snapshots: return to Avalonia Render Loop



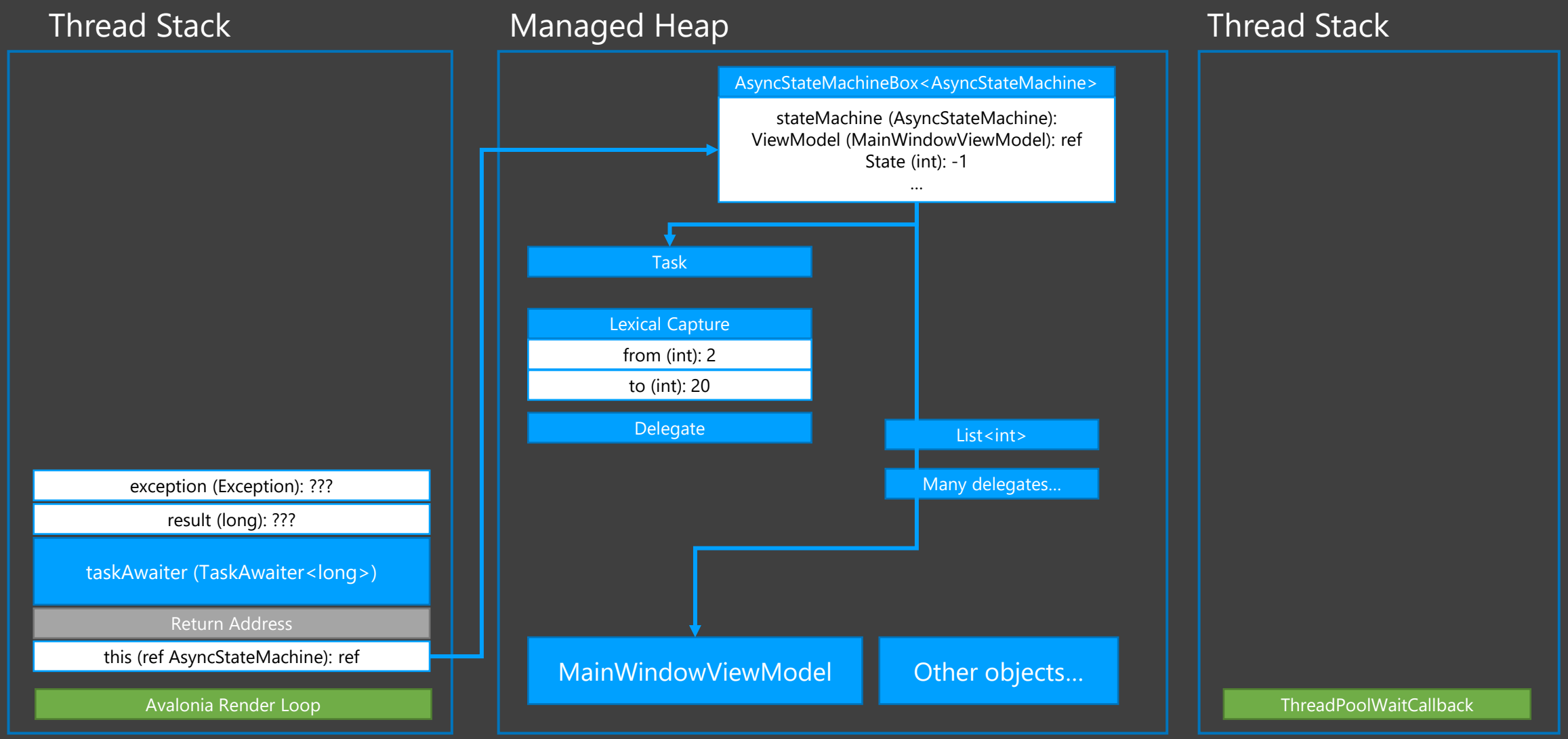
Memory Snapshots: progress on background thread



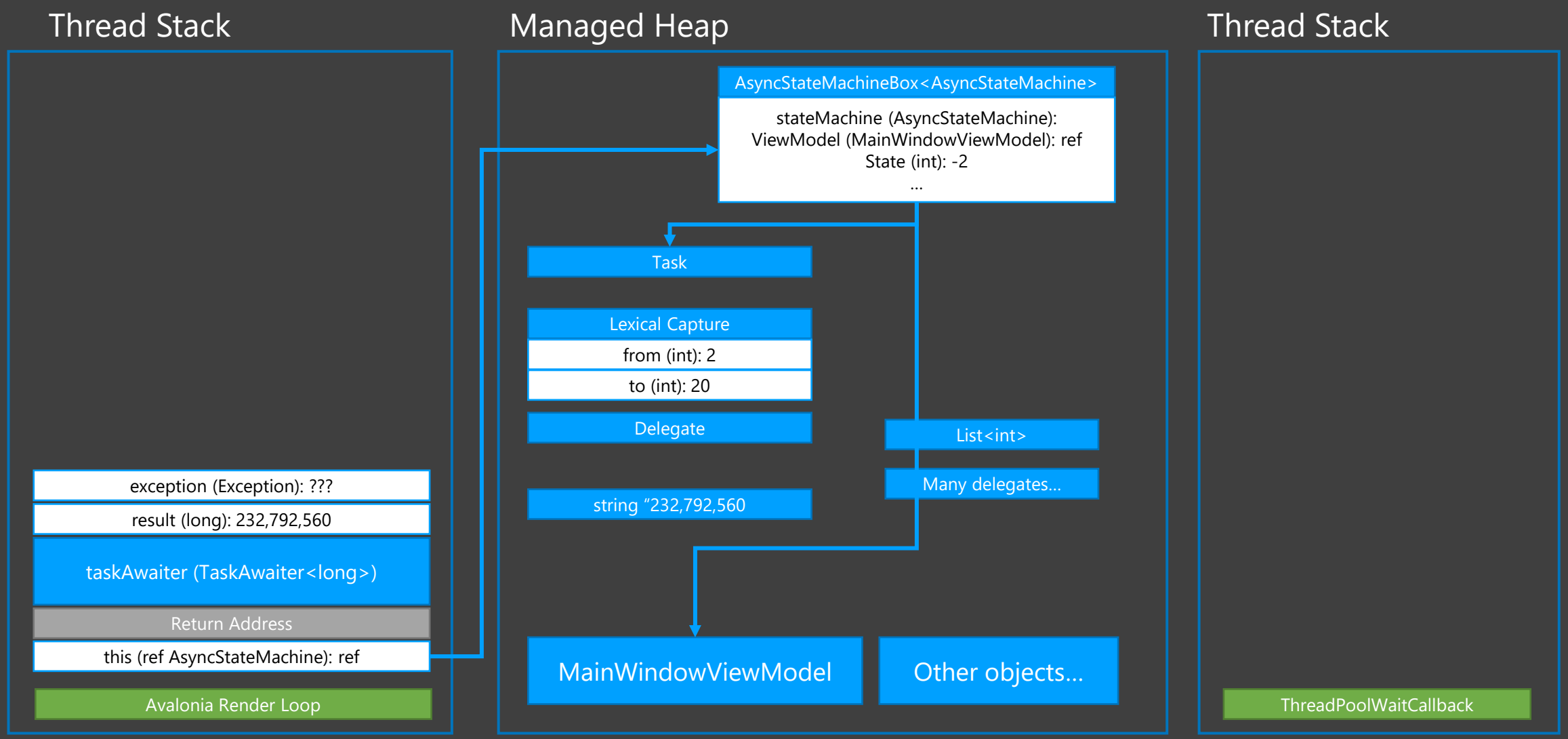
Memory Snapshots: after task is completed



Memory Snapshots: continuation started on UI thread



Memory Snapshots: at the end of MoveNext



Outro

What can we learn from all this?

We need new principles, esp. for young developers

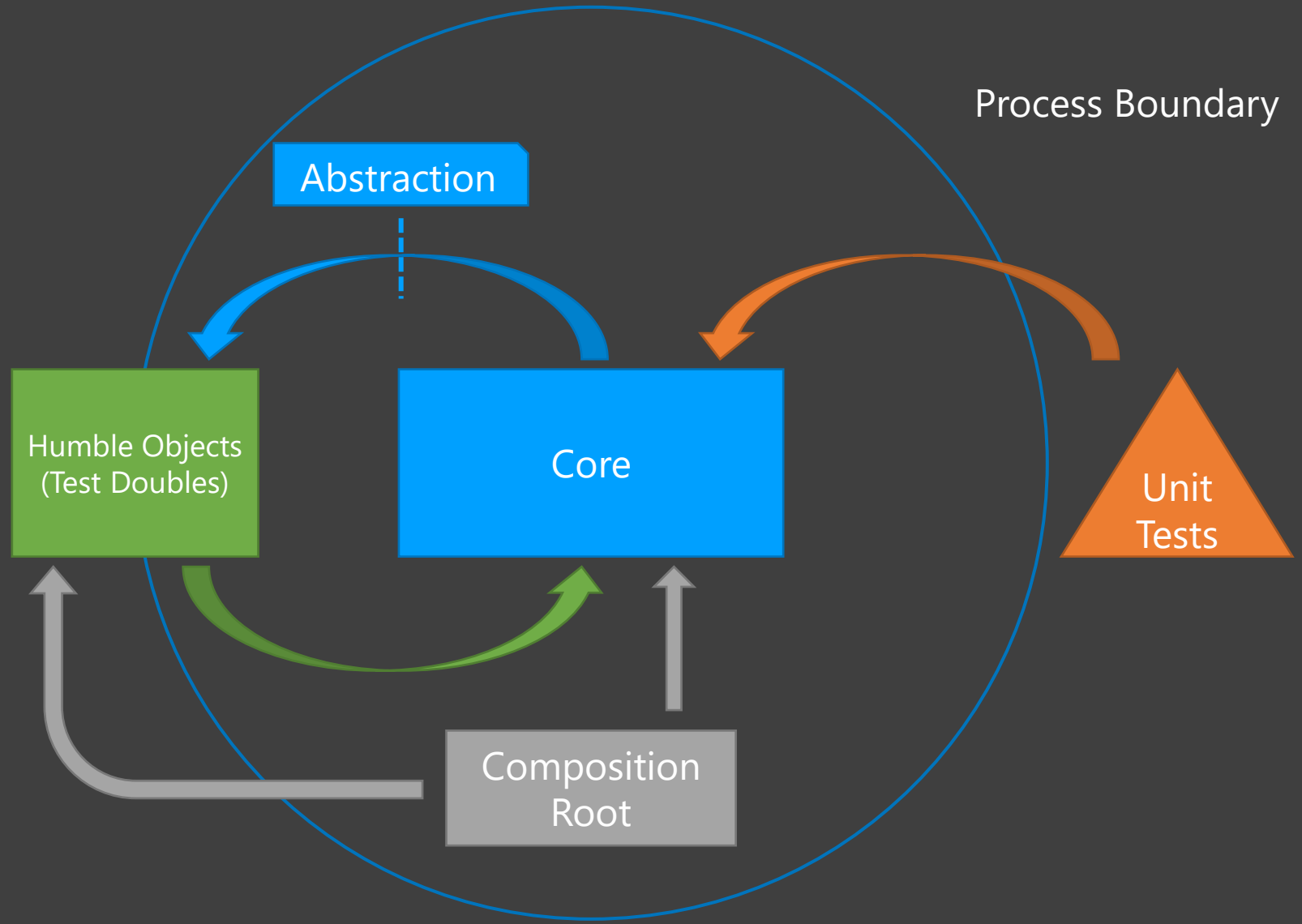
Learn the Internals (LTI)

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 way longer execution times. 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.

CHUC: Core – Humble Objects – Unit Tests – Composition Root



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
- Stephen Cleary: There is no thread
- Konrad Kokosa: Pro .NET Memory Management
- Mark Seemann: Dependency Injection in .NET

Thank you!

Got any questions? Let's have a drink!