The Evolution of Async Programming in C#

From Threads to Async/Await

Mühlehner & Tavolato GmbH

Agenda

Agenda

- **Traditional Threading** - Manual creation and management

Agenda

- **Traditional Threading** - Manual creation and management
- **ThreadPool** - Optimization of thread resources

Agenda

- **Traditional Threading** - Manual creation and management
- **ThreadPool** - Optimization of thread resources
- **BackgroundWorker** - Event-based UI-friendly pattern

Agenda

- **Traditional Threading** - Manual creation and management
- **ThreadPool** - Optimization of thread resources
- **BackgroundWorker** - Event-based UI-friendly pattern
- **Asynchronous Programming Model (APM)** - Begin/End pattern

Agenda

- **Traditional Threading** - Manual creation and management
- **ThreadPool** - Optimization of thread resources
- **BackgroundWorker** - Event-based UI-friendly pattern
- **Asynchronous Programming Model (APM)** - Begin/End pattern
- **Event-based Asynchronous Pattern (EAP)** - Event-driven async

Agenda

- **Traditional Threading** - Manual creation and management
- **ThreadPool** - Optimization of thread resources
- **BackgroundWorker** - Event-based UI-friendly pattern
- **Asynchronous Programming Model (APM)** - Begin/End pattern
- **Event-based Asynchronous Pattern (EAP)** - Event-driven async
- **Task Parallel Library (TPL)** - The Task abstraction

Agenda

- **Traditional Threading** - Manual creation and management
- **ThreadPool** - Optimization of thread resources
- **BackgroundWorker** - Event-based UI-friendly pattern
- **Asynchronous Programming Model (APM)** - Begin/End pattern
- **Event-based Asynchronous Pattern (EAP)** - Event-driven async
- **Task Parallel Library (TPL)** - The Task abstraction
- **Async/Await** - Revolutionary syntax transformation

Agenda

- **Traditional Threading** - Manual creation and management
- **ThreadPool** - Optimization of thread resources
- **BackgroundWorker** - Event-based UI-friendly pattern
- **Asynchronous Programming Model (APM)** - Begin/End pattern
- **Event-based Asynchronous Pattern (EAP)** - Event-driven async
- **Task Parallel Library (TPL)** - The Task abstraction
- **Async/Await** - Revolutionary syntax transformation
- **Modern C# Async Features** - Recent innovations

Why Asynchronous Programming?

Why Asynchronous Programming?

- **Responsiveness**: Keep UI thread free

Why Asynchronous Programming?

- **Responsiveness**: Keep UI thread free
- **Scalability**: Handle more concurrent operations

Why Asynchronous Programming?

- **Responsiveness**: Keep UI thread free
- **Scalability**: Handle more concurrent operations
- **Resource Efficiency**: Avoid blocking threads

Why Asynchronous Programming?

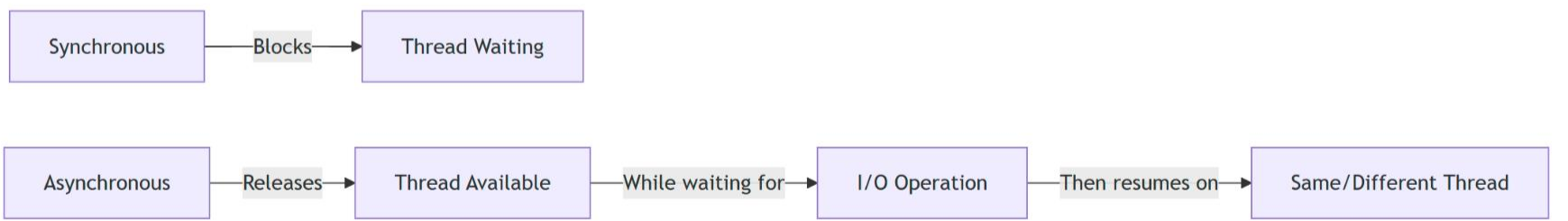
- **Responsiveness**: Keep UI thread free
- **Scalability**: Handle more concurrent operations
- **Resource Efficiency**: Avoid blocking threads
- **Performance**: Better utilization of multi-core processors

Why Asynchronous Programming?

- **Responsiveness**: Keep UI thread free
- **Scalability**: Handle more concurrent operations
- **Resource Efficiency**: Avoid blocking threads
- **Performance**: Better utilization of multi-core processors
- **I/O-bound Operations**: Efficiently wait for external resources

Why Asynchronous Programming?

- **Responsiveness**: Keep UI thread free
- **Scalability**: Handle more concurrent operations
- **Resource Efficiency**: Avoid blocking threads
- **Performance**: Better utilization of multi-core processors
- **I/O-bound Operations**: Efficiently wait for external resources



The Problem Space

The Problem Space

- **I/O-bound operations**: Network, disk, database

The Problem Space

- **I/O-bound operations**: Network, disk, database
- **CPU-bound operations**: Complex calculations

The Problem Space

- **I/O-bound operations**: Network, disk, database
- **CPU-bound operations**: Complex calculations
- **Concurrency vs Parallelism**

The Problem Space

- **I/O-bound operations**: Network, disk, database
- **CPU-bound operations**: Complex calculations
- **Concurrency vs Parallelism**
- **Coordination challenges**

The Problem Space

- **I/O-bound operations**: Network, disk, database
- **CPU-bound operations**: Complex calculations
- **Concurrency vs Parallelism**
- **Coordination challenges**
- **Error handling across threads**

The Problem Space

- **I/O-bound operations**: Network, disk, database
- **CPU-bound operations**: Complex calculations
- **Concurrency vs Parallelism**
- **Coordination challenges**
- **Error handling across threads**
- **Resource management**

The Problem Space

- **I/O-bound operations**: Network, disk, database
- **CPU-bound operations**: Complex calculations
- **Concurrency vs Parallelism**
- **Coordination challenges**
- **Error handling across threads**
- **Resource management**
- **Cancellation support**

1. Traditional Threading

```
1   Thread workerThread = new Thread(new ThreadStart(DoWo
2   workerThread.Start();
3
4   // Main thread continues execution
5   Console.WriteLine("Main thread working");
6
7   // Wait for worker thread to complete
8   workerThread.Join();
9
10  static void DoWork()
11  {
12      // Long-running operation
13      Thread.Sleep(1000);
14      Console.WriteLine("Worker thread completed");
15  }
```

1. Traditional Threading

- Manual thread creation with `System.Threading.Thread`

```
Thread workerThread = new Thread(new ThreadStart(DoWo
workerThread.Start();

// Main thread continues execution
Console.WriteLine("Main thread working");

// Wait for worker thread to complete
workerThread.Join();

static void DoWork()
{
    // Long-running operation
    Thread.Sleep(1000);
    Console.WriteLine("Worker thread completed");
}
```

1. Traditional Threading

- Manual thread creation with `System.Threading.Thread`
- Direct control but high complexity

```
Thread workerThread = new Thread(new ThreadStart(DoWo
workerThread.Start();

// Main thread continues execution
Console.WriteLine("Main thread working");

// Wait for worker thread to complete
workerThread.Join();

static void DoWork()
{
    // Long-running operation
    Thread.Sleep(1000);
    Console.WriteLine("Worker thread completed");
}
```

1. Traditional Threading

- Manual thread creation with `System.Threading.Thread`
- Direct control but high complexity
- Threads are **OS resources**

```
1   Thread workerThread = new Thread(new ThreadStart(DoWo
2   workerThread.Start();
3
4   // Main thread continues execution
5   Console.WriteLine("Main thread working");
6
7   // Wait for worker thread to complete
8   workerThread.Join();
9
10  static void DoWork()
11  {
12      // Long-running operation
13      Thread.Sleep(1000);
14      Console.WriteLine("Worker thread completed");
15  }
```

1. Traditional Threading

- Manual thread creation with `System.Threading.Thread`
- Direct control but high complexity
- Threads are **OS resources**
- Fixed overhead (~1MB stack)

```
1   Thread workerThread = new Thread(new ThreadStart(DoWo
2   workerThread.Start();
3
4   // Main thread continues execution
5   Console.WriteLine("Main thread working");
6
7   // Wait for worker thread to complete
8   workerThread.Join();
9
10  static void DoWork()
11  {
12      // Long-running operation
13      Thread.Sleep(1000);
14      Console.WriteLine("Worker thread completed");
15  }
```

1. Traditional Threading

- Manual thread creation with `System.Threading.Thread`
- Direct control but high complexity
- Threads are **OS resources**
- Fixed overhead (~1MB stack)
- No built-in way to return values

```
1   Thread workerThread = new Thread(new ThreadStart(DoWo
2   workerThread.Start();
3
4   // Main thread continues execution
5   Console.WriteLine("Main thread working");
6
7   // Wait for worker thread to complete
8   workerThread.Join();
9
10  static void DoWork()
11  {
12      // Long-running operation
13      Thread.Sleep(1000);
14      Console.WriteLine("Worker thread completed");
15  }
```

1. Traditional Threading

- Manual thread creation with `System.Threading.Thread`
- Direct control but high complexity
- Threads are **OS resources**
- Fixed overhead (~1MB stack)
- No built-in way to return values
- Complex error handling

```
Thread workerThread = new Thread(new ThreadStart(DoWo
workerThread.Start();

// Main thread continues execution
Console.WriteLine("Main thread working");

// Wait for worker thread to complete
workerThread.Join();

static void DoWork()
{
    // Long-running operation
    Thread.Sleep(1000);
    Console.WriteLine("Worker thread completed");
}
```

How Threading Works Under the Hood

How Threading Works Under the Hood

- Each thread maps to an OS thread

How Threading Works Under the Hood

- Each thread maps to an OS thread
- Thread scheduling is handled by OS

How Threading Works Under the Hood

- Each thread maps to an OS thread
- Thread scheduling is handled by OS
- Context switching is expensive

How Threading Works Under the Hood

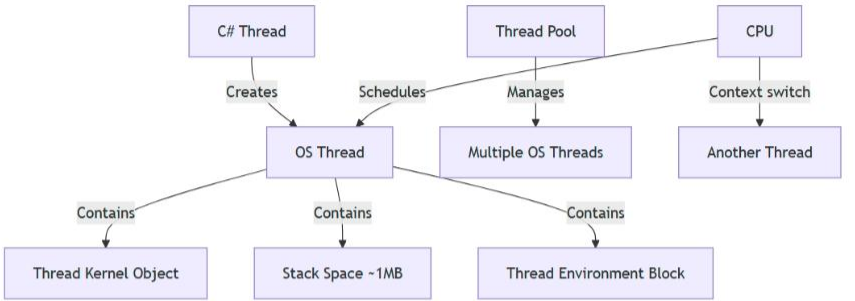
- Each thread maps to an OS thread
- Thread scheduling is handled by OS
- Context switching is expensive
- Thread creation is resource-intensive

How Threading Works Under the Hood

- Each thread maps to an OS thread
- Thread scheduling is handled by OS
- Context switching is expensive
- Thread creation is resource-intensive
- Thread overhead includes:
  - Thread kernel object
  - Thread environment block
  - User and kernel stacks
  - DLL thread-attach notifications

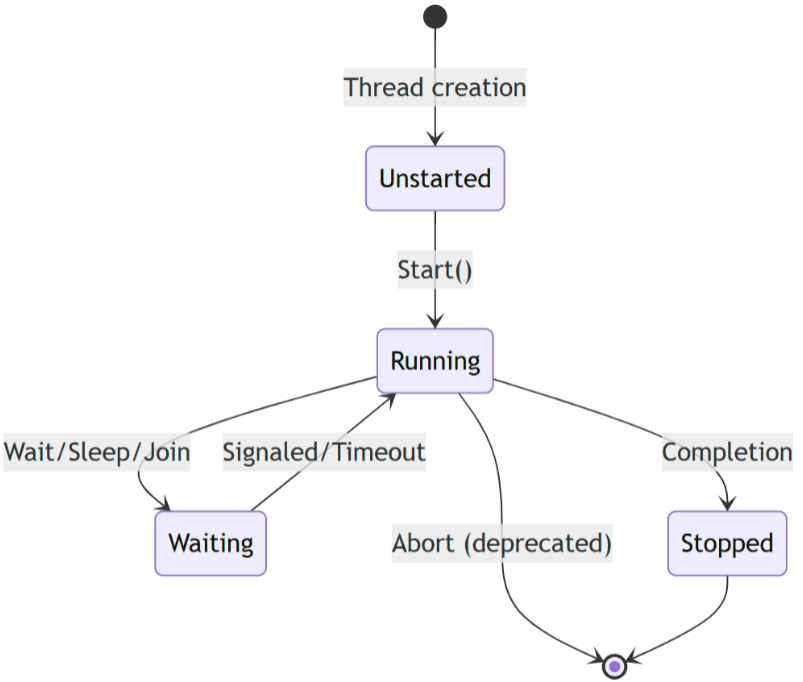
How Threading Works Under the Hood

- Each thread maps to an OS thread
- Thread scheduling is handled by OS
- Context switching is expensive
- Thread creation is resource-intensive
- Thread overhead includes:
  - Thread kernel object
  - Thread environment block
  - User and kernel stacks
  - DLL thread-attach notifications



Thread States and Transitions

Thread States and Transitions



Mühlehner & Tavolato GmbH

Thread Synchronization

- Mutex
- Semaphore
- Monitor (lock)
- ManualResetEvent
- AutoResetEvent
- ReaderWriterLock

Thread Synchronization

- Mutex
- Semaphore
- Monitor (lock)
- ManualResetEvent
- AutoResetEvent
- ReaderWriterLock

Thread Scheduling

- Thread.Priority
- Preemptive multitasking
- Time slicing
- Thread quantum
- Thread affinity

2. ThreadPool Introduction

```
// Queue a work item to the thread pool
ThreadPool.QueueUserWorkItem(
    new WaitCallback(state =>
    {
        Console.WriteLine($"Working on: {state}");
        // Do work here
    }),
    "Task Data"
);

// Main thread continues execution
Console.WriteLine("Main thread continues...");
```

2. ThreadPool Introduction

- Reuses threads to reduce overhead

```
// Queue a work item to the thread pool
ThreadPool.QueueUserWorkItem(
    new WaitCallback(state =>
    {
        Console.WriteLine($"Working on: {state}");
        // Do work here
    }),
    "Task Data"
);

// Main thread continues execution
Console.WriteLine("Main thread continues...");
```

2. ThreadPool Introduction

- Reuses threads to reduce overhead
- Managed by the CLR

```
// Queue a work item to the thread pool
ThreadPool.QueueUserWorkItem(
    new WaitCallback(state =>
    {
        Console.WriteLine($"Working on: {state}");
        // Do work here
    }),
    "Task Data"
);

// Main thread continues execution
Console.WriteLine("Main thread continues...");
```

2. ThreadPool Introduction

- Reuses threads to reduce overhead
- Managed by the CLR
- Optimized for many short operations

```
// Queue a work item to the thread pool
ThreadPool.QueueUserWorkItem(
    new WaitCallback(state =>
    {
        Console.WriteLine($"Working on: {state}");
        // Do work here
    }),
    "Task Data"
);

// Main thread continues execution
Console.WriteLine("Main thread continues...");
```

2. ThreadPool Introduction

- Reuses threads to reduce overhead
- Managed by the CLR
- Optimized for many short operations
- Limited control over scheduling

```
// Queue a work item to the thread pool
ThreadPool.QueueUserWorkItem(
    new WaitCallback(state =>
    {
        Console.WriteLine($"Working on: {state}");
        // Do work here
    }),
    "Task Data"
);

// Main thread continues execution
Console.WriteLine("Main thread continues...");
```

2. ThreadPool Introduction

- Reuses threads to reduce overhead
- Managed by the CLR
- Optimized for many short operations
- Limited control over scheduling
- Cannot easily track specific work items

```
// Queue a work item to the thread pool
ThreadPool.QueueUserWorkItem(
    new WaitCallback(state =>
    {
        Console.WriteLine($"Working on: {state}");
        // Do work here
    }),
    "Task Data"
);

// Main thread continues execution
Console.WriteLine("Main thread continues...");
```

2. ThreadPool Introduction

- Reuses threads to reduce overhead
- Managed by the CLR
- Optimized for many short operations
- Limited control over scheduling
- Cannot easily track specific work items
- No built-in cancellation or progress

```
// Queue a work item to the thread pool
ThreadPool.QueueUserWorkItem(
    new WaitCallback(state =>
    {
        Console.WriteLine($"Working on: {state}");
        // Do work here
    }),
    "Task Data"
);

// Main thread continues execution
Console.WriteLine("Main thread continues...");
```

ThreadPool Under the Hood

ThreadPool Under the Hood

- Maintains worker threads based on demand

ThreadPool Under the Hood

- Maintains worker threads based on demand
- Uses a global queue and per-processor queues

ThreadPool Under the Hood

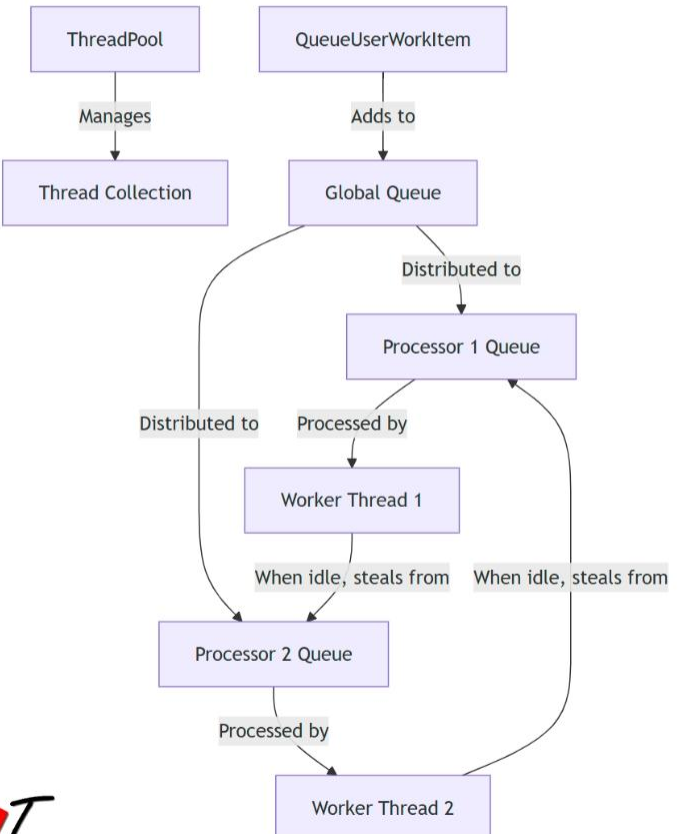
- Maintains worker threads based on demand
- Uses a global queue and per-processor queues
- Worker threads automatically adjust:
  - Minimum threads per processor
  - Dynamic growth based on demand
  - Thread injection algorithm
  - Hill-climbing algorithm for optimization

ThreadPool Under the Hood

- Maintains worker threads based on demand
- Uses a global queue and per-processor queues
- Worker threads automatically adjust:
  - Minimum threads per processor
  - Dynamic growth based on demand
  - Thread injection algorithm
  - Hill-climbing algorithm for optimization
- Uses work-stealing for load balancing

ThreadPool Under the Hood

- Maintains worker threads based on demand
- Uses a global queue and per-processor queues
- Worker threads automatically adjust:
  - Minimum threads per processor
  - Dynamic growth based on demand
  - Thread injection algorithm
  - Hill-climbing algorithm for optimization
- Uses work-stealing for load balancing



ThreadPool vs Manual Threads

ThreadPool vs Manual Threads

- **ThreadPool**: Reuses threads, efficient for many small tasks
- **Manual Threads**: Better for long-running, dedicated operations

ThreadPool vs Manual Threads

- **ThreadPool**: Reuses threads, efficient for many small tasks
- **Manual Threads**: Better for long-running, dedicated operations

Feature	Manual Threads	ThreadPool
Thread creation overhead	High (every time)	Low (amortized)
Control over threads	Complete	Limited
Lifetime management	Manual	Automatic
Foreground/Background	Configurable	Always background
Priority control	Supported	Not supported
Thread naming	Supported	Not supported

3. BackgroundWorker

```
1   BackgroundWorker worker = new BackgroundWorker();
2   worker.WorkerReportsProgress = true;
3   worker.WorkerSupportsCancellation = true;
4
5   // Set up event handlers
6   worker.DoWork += Worker_DoWork;
7   worker.ProgressChanged += Worker_ProgressChanged;
8   worker.RunWorkerCompleted += Worker_RunWorkerComplete
9
10  // Start the worker
11  worker.RunWorkerAsync("Parameter");
12
13  // Later, to cancel:
14  worker.CancelAsync();
```

3. BackgroundWorker

- Designed for UI applications

```
1   BackgroundWorker worker = new BackgroundWorker();
2   worker.WorkerReportsProgress = true;
3   worker.WorkerSupportsCancellation = true;
4
5   // Set up event handlers
6   worker.DoWork += Worker_DoWork;
7   worker.ProgressChanged += Worker_ProgressChanged;
8   worker.RunWorkerCompleted += Worker_RunWorkerComplete
9
10  // Start the worker
11  worker.RunWorkerAsync("Parameter");
12
13  // Later, to cancel:
14  worker.CancelAsync();
```

3. BackgroundWorker

- Designed for UI applications
- Event-based, UI-thread friendly

```
BackgroundWorker worker = new BackgroundWorker();
worker.WorkerReportsProgress = true;
worker.WorkerSupportsCancellation = true;

// Set up event handlers
worker.DoWork += Worker_DoWork;
worker.ProgressChanged += Worker_ProgressChanged;
worker.RunWorkerCompleted += Worker_RunWorkerComplete

// Start the worker
worker.RunWorkerAsync("Parameter");

// Later, to cancel:
worker.CancelAsync();
```

3. BackgroundWorker

- Designed for UI applications
- Event-based, UI-thread friendly
- Built-in progress reporting

```
1   BackgroundWorker worker = new BackgroundWorker();
2   worker.WorkerReportsProgress = true;
3   worker.WorkerSupportsCancellation = true;
4
5   // Set up event handlers
6   worker.DoWork += Worker_DoWork;
7   worker.ProgressChanged += Worker_ProgressChanged;
8   worker.RunWorkerCompleted += Worker_RunWorkerComplete
9
10  // Start the worker
11  worker.RunWorkerAsync("Parameter");
12
13  // Later, to cancel:
14  worker.CancelAsync();
```

3. BackgroundWorker

- Designed for UI applications
- Event-based, UI-thread friendly
- Built-in progress reporting
- Cancellation support

```
1   BackgroundWorker worker = new BackgroundWorker();
2   worker.WorkerReportsProgress = true;
3   worker.WorkerSupportsCancellation = true;
4
5   // Set up event handlers
6   worker.DoWork += Worker_DoWork;
7   worker.ProgressChanged += Worker_ProgressChanged;
8   worker.RunWorkerCompleted += Worker_RunWorkerComplete
9
10  // Start the worker
11  worker.RunWorkerAsync("Parameter");
12
13  // Later, to cancel:
14  worker.CancelAsync();
```

3. BackgroundWorker

- Designed for UI applications
- Event-based, UI-thread friendly
- Built-in progress reporting
- Cancellation support
- Automatic marshaling to UI thread

```
1   BackgroundWorker worker = new BackgroundWorker();
2   worker.WorkerReportsProgress = true;
3   worker.WorkerSupportsCancellation = true;
4
5   // Set up event handlers
6   worker.DoWork += Worker_DoWork;
7   worker.ProgressChanged += Worker_ProgressChanged;
8   worker.RunWorkerCompleted += Worker_RunWorkerComplete
9
10  // Start the worker
11  worker.RunWorkerAsync("Parameter");
12
13  // Later, to cancel:
14  worker.CancelAsync();
```

3. BackgroundWorker

- Designed for UI applications
- Event-based, UI-thread friendly
- Built-in progress reporting
- Cancellation support
- Automatic marshaling to UI thread
- Simpler error handling

```
1   BackgroundWorker worker = new BackgroundWorker();
2   worker.WorkerReportsProgress = true;
3   worker.WorkerSupportsCancellation = true;
4
5   // Set up event handlers
6   worker.DoWork += Worker_DoWork;
7   worker.ProgressChanged += Worker_ProgressChanged;
8   worker.RunWorkerCompleted += Worker_RunWorkerComplete
9
10  // Start the worker
11  worker.RunWorkerAsync("Parameter");
12
13  // Later, to cancel:
14  worker.CancelAsync();
```

BackgroundWorker Under the Hood

Mühlehner & Tavolato GmbH

BackgroundWorker Under the Hood

- Uses ThreadPool internally

BackgroundWorker Under the Hood

- Uses ThreadPool internally
- Captures SynchronizationContext to:
  - Marshal events back to UI thread
  - Ensure thread safety for UI updates

BackgroundWorker Under the Hood

- Uses ThreadPool internally
- Captures SynchronizationContext to:
  - Marshal events back to UI thread
  - Ensure thread safety for UI updates
- Manages state transitions internally

BackgroundWorker Under the Hood

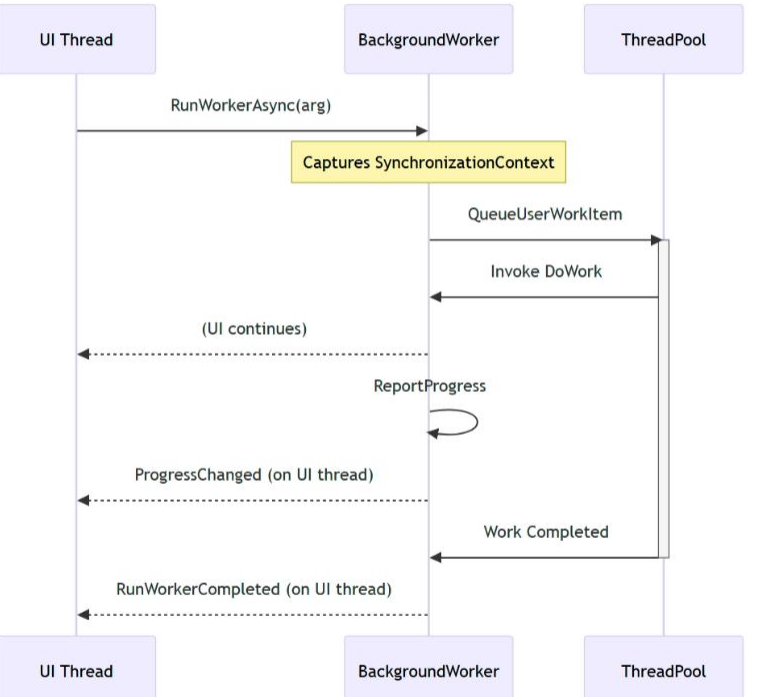
- Uses ThreadPool internally
- Captures SynchronizationContext to:
  - Marshal events back to UI thread
  - Ensure thread safety for UI updates
- Manages state transitions internally
- Wraps exceptions in RunWorkerCompletedEventArgs

BackgroundWorker Under the Hood

- Uses ThreadPool internally
- Captures SynchronizationContext to:
  - Marshal events back to UI thread
  - Ensure thread safety for UI updates
- Manages state transitions internally
- Wraps exceptions in RunWorkerCompletedEventArgs
- Provides thread safety via event pattern

BackgroundWorker Under the Hood

- Uses ThreadPool internally
- Captures SynchronizationContext to:
  - Marshal events back to UI thread
  - Ensure thread safety for UI updates
- Manages state transitions internally
- Wraps exceptions in RunWorkerCompletedEventArgs
- Provides thread safety via event pattern



4. Asynchronous Programming Model (APM)

```
// Begin the asynchronous operation
IAsyncResult asyncResult = webRequest.BeginGetRespons
    new AsyncCallback(GetResponseCallback),
    webRequest);

// Optionally wait for completion
asyncResult.AsyncWaitHandle.WaitOne();

// Callback function
void GetResponseCallback(IAsyncResult ar)
{
    WebRequest request = (WebRequest)ar.AsyncState;

    // End the async operation (get result)
    WebResponse response = request.EndGetResponse(ar)

    // Process the response
    // ...
}
```

4. Asynchronous Programming Model (APM)

- Pattern with BeginXxx and EndXxx methods

```
// Begin the asynchronous operation
IAsyncResult asyncResult = webRequest.BeginGetRespons
    new AsyncCallback(GetResponseCallback),
    webRequest);

// Optionally wait for completion
asyncResult.AsyncWaitHandle.WaitOne();

// Callback function
void GetResponseCallback(IAsyncResult ar)
{
    WebRequest request = (WebRequest)ar.AsyncState;

    // End the async operation (get result)
    WebResponse response = request.EndGetResponse(ar)

    // Process the response
    // ...
}
```

4. Asynchronous Programming Model (APM)

- Pattern with BeginXxx and EndXxx methods
- Based on IAsyncResult interface

```
1   // Begin the asynchronous operation
2   IAsyncResult asyncResult = webRequest.BeginGetRespons
3       new AsyncCallback(GetResponseCallback),
4       webRequest);
5
6   // Optionally wait for completion
7   asyncResult.AsyncWaitHandle.WaitOne();
8
9   // Callback function
10  void GetResponseCallback(IAsyncResult ar)
11  {
12      WebRequest request = (WebRequest)ar.AsyncState;
13
14      // End the async operation (get result)
15      WebResponse response = request.EndGetResponse(ar
16
17      // Process the response
18      // ...
19  }
```

4. Asynchronous Programming Model (APM)

- Pattern with BeginXxx and EndXxx methods
- Based on IAsyncResult interface
- Widely used throughout .NET Framework

```
// Begin the asynchronous operation
IAsyncResult asyncResult = webRequest.BeginGetRespons
    new AsyncCallback(GetResponseCallback),
    webRequest);

// Optionally wait for completion
asyncResult.AsyncWaitHandle.WaitOne();

// Callback function
void GetResponseCallback(IAsyncResult ar)
{
    WebRequest request = (WebRequest)ar.AsyncState;

    // End the async operation (get result)
    WebResponse response = request.EndGetResponse(ar)

    // Process the response
    // ...
}
```

4. Asynchronous Programming Model (APM)

- Pattern with BeginXxx and EndXxx methods
- Based on IAsyncResult interface
- Widely used throughout .NET Framework
- Supports both callback and wait-based programming

```
// Begin the asynchronous operation
IAsyncResult asyncResult = webRequest.BeginGetRespons
    new AsyncCallback(GetResponseCallback),
    webRequest);

// Optionally wait for completion
asyncResult.AsyncWaitHandle.WaitOne();

// Callback function
void GetResponseCallback(IAsyncResult ar)
{
    WebRequest request = (WebRequest)ar.AsyncState;

    // End the async operation (get result)
    WebResponse response = request.EndGetResponse(ar)

    // Process the response
    // ...
}
```

4. Asynchronous Programming Model (APM)

- Pattern with BeginXxx and EndXxx methods
- Based on IAsyncResult interface
- Widely used throughout .NET Framework
- Supports both callback and wait-based programming
- Often implemented with ThreadPool

```
1   // Begin the asynchronous operation
2   IAsyncResult asyncResult = webRequest.BeginGetRespons
3       new AsyncCallback(GetResponseCallback),
4       webRequest);
5
6   // Optionally wait for completion
7   asyncResult.AsyncWaitHandle.WaitOne();
8
9   // Callback function
10  void GetResponseCallback(IAsyncResult ar)
11  {
12      WebRequest request = (WebRequest)ar.AsyncState;
13
14      // End the async operation (get result)
15      WebResponse response = request.EndGetResponse(ar
16
17      // Process the response
18      // ...
19  }
```

4. Asynchronous Programming Model (APM)

- Pattern with BeginXxx and EndXxx methods
- Based on IAsyncResult interface
- Widely used throughout .NET Framework
- Supports both callback and wait-based programming
- Often implemented with ThreadPool
- Foundation of many .NET 2.0 async APIs

```
// Begin the asynchronous operation
IAsyncResult asyncResult = webRequest.BeginGetRespons
    new AsyncCallback(GetResponseCallback),
    webRequest);

// Optionally wait for completion
asyncResult.AsyncWaitHandle.WaitOne();

// Callback function
void GetResponseCallback(IAsyncResult ar)
{
    WebRequest request = (WebRequest)ar.AsyncState;

    // End the async operation (get result)
    WebResponse response = request.EndGetResponse(ar)

    // Process the response
    // ...
}
```

APM Under the Hood: IAsyncResult

APM Under the Hood: IAsyncResult

```
1   public interface IAsyncResult
2   {
3       // Gets a user-defined object that
4       // qualifies or contains information
5       // about an async operation
6       object AsyncState { get; }
7
8       // Gets a WaitHandle that can be used
9       // to wait for the async operation
10      WaitHandle AsyncWaitHandle { get; }
11
12      // Gets a value indicating whether the
13      // async operation completed synchronously
14      bool CompletedSynchronously { get; }
15
16      // Gets a value indicating whether the
17      // async operation has completed
18      bool IsCompleted { get; }
19  }
```

Mühlehner & Tavolato GmbH

APM Under the Hood: IAsyncResult

```
1   public interface IAsyncResult
2   {
3       // Gets a user-defined object that
4       // qualifies or contains information
5       // about an async operation
6       object AsyncState { get; }
7
8       // Gets a WaitHandle that can be used
9       // to wait for the async operation
10      WaitHandle AsyncWaitHandle { get; }
11
12      // Gets a value indicating whether the
13      // async operation completed synchronously
14      bool CompletedSynchronously { get; }
15
16      // Gets a value indicating whether the
17      // async operation has completed
18      bool IsCompleted { get; }
19  }
```

EndXxx Method Mechanics

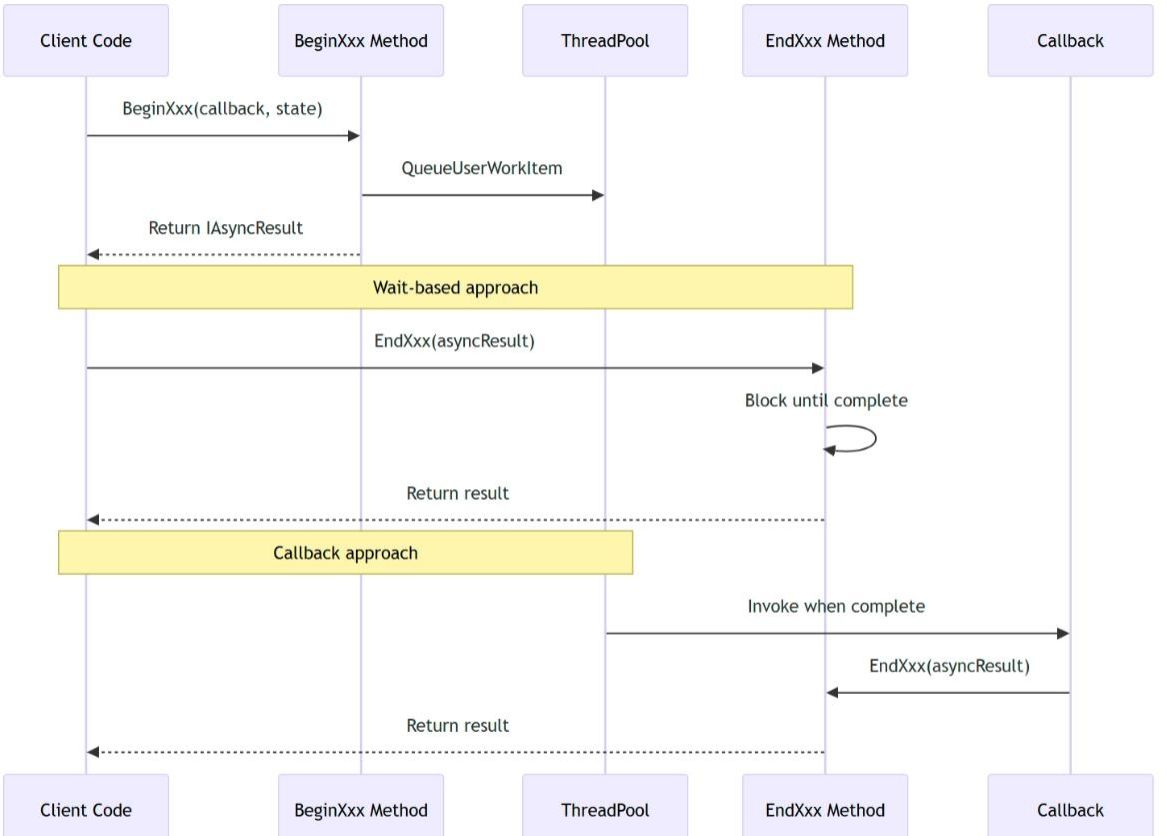
- Blocks until operation completes (if not already)
- Retrieves result (or throws exception)
- Releases resources used by async operation
- Must be called exactly once per BeginXxx call

Common Implementation Pattern

- ThreadPool-based implementation
- State machine pattern
- WaitHandle signaling
- Exception marshaling from worker thread

APM Execution Flow

APM Execution Flow



Mühlehner & Tavolato GmbH

Mühlehner & Tavolato GmbH

APM in .NET Framework

- Stream.BeginRead/EndRead
- WebRequest.BeginGetResponse/EndGetResponse
- Socket.BeginConnect/EndConnect
- FileStream.BeginWrite/EndWrite
- SqlCommand.BeginExecuteReader/EndExecuteReader

5. Event-based Asynchronous Pattern (EAP)

```
WebClient client = new WebClient();

// Set up the event handlers
client.DownloadStringCompleted += (s, e) ⇒
{
    if (e.Cancelled)
        Console.WriteLine("Download cancelled");
    else if (e.Error ≠ null)
        Console.WriteLine($"Error: {e.Error.Message}");
    else
        Console.WriteLine($"Result: {e.Result}");
};

client.DownloadProgressChanged += (s, e) ⇒
{
    Console.WriteLine($"Progress: {e.ProgressPercentage}%");
};

// Start the asynchronous operation
client.DownloadStringAsync(new Uri("http://example.com"));

// Later, to cancel:
client.CancelAsync();
```

Mühlehner & Tavolato GmbH

5. Event-based Asynchronous Pattern (EAP)

- Simplified event-based model

```
WebClient client = new WebClient();

// Set up the event handlers
client.DownloadStringCompleted += (s, e) ⇒
{
    if (e.Cancelled)
        Console.WriteLine("Download cancelled");
    else if (e.Error ≠ null)
        Console.WriteLine($"Error: {e.Error.Message}");
    else
        Console.WriteLine($"Result: {e.Result}");
};

client.DownloadProgressChanged += (s, e) ⇒
{
    Console.WriteLine($"Progress: {e.ProgressPercentage}%");
};

// Start the asynchronous operation
client.DownloadStringAsync(new Uri("http://example.com"));

// Later, to cancel:
client.CancelAsync();
```

5. Event-based Asynchronous Pattern (EAP)

- Simplified event-based model
- MethodNameAsync methods

```
WebClient client = new WebClient();

// Set up the event handlers
client.DownloadStringCompleted += (s, e) ⇒
{
    if (e.Cancelled)
        Console.WriteLine("Download cancelled");
    else if (e.Error ≠ null)
        Console.WriteLine($"Error: {e.Error.Message}");
    else
        Console.WriteLine($"Result: {e.Result}");
};

client.DownloadProgressChanged += (s, e) ⇒
{
    Console.WriteLine($"Progress: {e.ProgressPercentage}%");
};

// Start the asynchronous operation
client.DownloadStringAsync(new Uri("http://example.com"));

// Later, to cancel:
client.CancelAsync();
```

5. Event-based Asynchronous Pattern (EAP)

- Simplified event-based model
- MethodNameAsync methods
- MethodNameCompleted events

```
WebClient client = new WebClient();

// Set up the event handlers
client.DownloadStringCompleted += (s, e) ⇒
{
    if (e.Cancelled)
        Console.WriteLine("Download cancelled");
    else if (e.Error ≠ null)
        Console.WriteLine($"Error: {e.Error.Message}");
    else
        Console.WriteLine($"Result: {e.Result}");
};

client.DownloadProgressChanged += (s, e) ⇒
{
    Console.WriteLine($"Progress: {e.ProgressPercentage}%");
};

// Start the asynchronous operation
client.DownloadStringAsync(new Uri("http://example.com"));

// Later, to cancel:
client.CancelAsync();
```

5. Event-based Asynchronous Pattern (EAP)

- Simplified event-based model
- MethodNameAsync methods
- MethodNameCompleted events
- Progress and cancellation support

```
WebClient client = new WebClient();

// Set up the event handlers
client.DownloadStringCompleted += (s, e) ⇒
{
    if (e.Cancelled)
        Console.WriteLine("Download cancelled");
    else if (e.Error ≠ null)
        Console.WriteLine($"Error: {e.Error.Message}");
    else
        Console.WriteLine($"Result: {e.Result}");
};

client.DownloadProgressChanged += (s, e) ⇒
{
    Console.WriteLine($"Progress: {e.ProgressPercentage}%");
};

// Start the asynchronous operation
client.DownloadStringAsync(new Uri("http://example.com"));

// Later, to cancel:
client.CancelAsync();
```

5. Event-based Asynchronous Pattern (EAP)

- Simplified event-based model
- MethodNameAsync methods
- MethodNameCompleted events
- Progress and cancellation support
- Based on APM or ThreadPool internally

```
WebClient client = new WebClient();

// Set up the event handlers
client.DownloadStringCompleted += (s, e) ⇒
{
    if (e.Cancelled)
        Console.WriteLine("Download cancelled");
    else if (e.Error ≠ null)
        Console.WriteLine($"Error: {e.Error.Message}");
    else
        Console.WriteLine($"Result: {e.Result}");
};

client.DownloadProgressChanged += (s, e) ⇒
{
    Console.WriteLine($"Progress: {e.ProgressPercentage}%");
};

// Start the asynchronous operation
client.DownloadStringAsync(new Uri("http://example.com"));

// Later, to cancel:
client.CancelAsync();
```

5. Event-based Asynchronous Pattern (EAP)

- Simplified event-based model
- MethodNameAsync methods
- MethodNameCompleted events
- Progress and cancellation support
- Based on APM or ThreadPool internally
- Better for UI programming than APM

```
WebClient client = new WebClient();

// Set up the event handlers
client.DownloadStringCompleted += (s, e) ⇒
{
    if (e.Cancelled)
        Console.WriteLine("Download cancelled");
    else if (e.Error ≠ null)
        Console.WriteLine($"Error: {e.Error.Message}");
    else
        Console.WriteLine($"Result: {e.Result}");
};

client.DownloadProgressChanged += (s, e) ⇒
{
    Console.WriteLine($"Progress: {e.ProgressPercentage}%");
};

// Start the asynchronous operation
client.DownloadStringAsync(new Uri("http://example.com"));

// Later, to cancel:
client.CancelAsync();
```

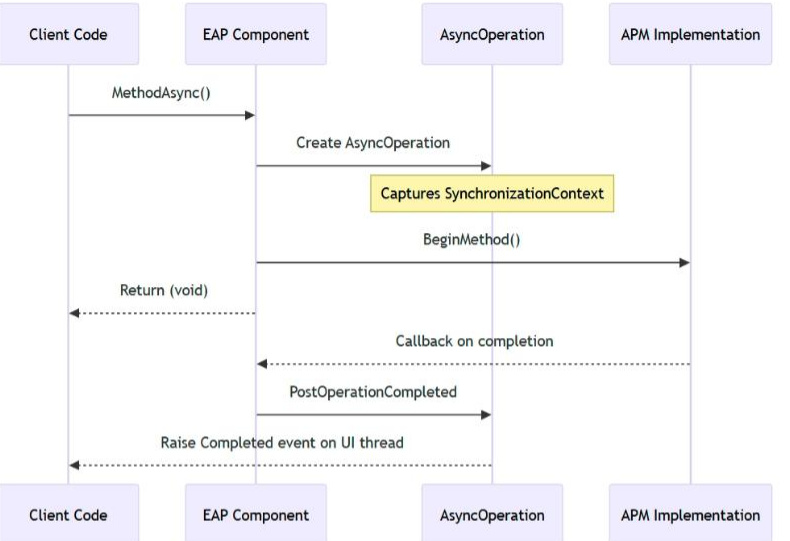
5. Event-based Asynchronous Pattern (EAP)

- Simplified event-based model
- MethodNameAsync methods
- MethodNameCompleted events
- Progress and cancellation support
- Based on APM or ThreadPool internally
- Better for UI programming than APM
- Used in .NET 2.0 and 3.5 components

```
WebClient client = new WebClient();

// Set up the event handlers
client.DownloadStringCompleted += (s, e) ⇒
{
    if (e.Cancelled)
        Console.WriteLine("Download cancelled");
    else if (e.Error ≠ null)
        Console.WriteLine($"Error: {e.Error.Message}");
    else
        Console.WriteLine($"Result: {e.Result}");
};

client.DownloadProgressChanged += (s, e) ⇒
{
    Console.WriteLine($"Progress: {e.ProgressPercentage}%");
};

// Start the asynchronous operation
client.DownloadStringAsync(new Uri("http://example.com"));

// Later, to cancel:
client.CancelAsync();
```

EAP Under the Hood

- Often implemented using APM pattern internally
- Captures SynchronizationContext like BackgroundWorker
- Encapsulates AsyncOperation class to manage threading
- Uses a state machine to track operation state
- Handles exception wrapping and marshaling
- Packages results in event args objects



6. Task Parallel Library (TPL)

6. Task Parallel Library (TPL)

- Introduced in .NET 4.0

6. Task Parallel Library (TPL)

- Introduced in .NET 4.0
- Task as a unit of asynchronous work

6. Task Parallel Library (TPL)

- Introduced in .NET 4.0
- Task as a unit of asynchronous work
- `Task<TResult>` for operations with return values

6. Task Parallel Library (TPL)

- Introduced in .NET 4.0
- Task as a unit of asynchronous work
- `Task<TResult>` for operations with return values
- Rich API for composition and synchronization

6. Task Parallel Library (TPL)

- Introduced in .NET 4.0
- Task as a unit of asynchronous work
- `Task<TResult>` for operations with return values
- Rich API for composition and synchronization
- Support for cancellation and continuations

6. Task Parallel Library (TPL)

- Introduced in .NET 4.0
- Task as a unit of asynchronous work
- `Task<TResult>` for operations with return values
- Rich API for composition and synchronization
- Support for cancellation and continuations
- Exception handling via AggregateException

6. Task Parallel Library (TPL)

- Introduced in .NET 4.0
- Task as a unit of asynchronous work
- `Task<TResult>` for operations with return values
- Rich API for composition and synchronization
- Support for cancellation and continuations
- Exception handling via AggregateException
- Foundation for async/await

```
// Create and start a task
Task<int> task = Task.Run(() ⇒
{
    // Perform computation
    return ComputeValue();
});

// Add a continuation
Task<string> continuation = task.ContinueWith(t ⇒
{
    // Process the result
    return $"Result: {t.Result}";
});

// Wait for the result
string result = continuation.Result;

// Create task completion source
var tcs = new TaskCompletionSource<int>();
// Later: tcs.SetResult(42);
```

Task Under the Hood

Task Under the Hood

- Task is a promise/future abstraction

Task Under the Hood

- Task is a promise/future abstraction
- Task != Thread (important distinction)

Task Under the Hood

- Task is a promise/future abstraction
- Task != Thread (important distinction)
- TaskScheduler determines execution strategy
  - Default scheduler uses ThreadPool
  - SynchronizationContextTaskScheduler for UI

Task Under the Hood

- Task is a promise/future abstraction
- Task != Thread (important distinction)
- TaskScheduler determines execution strategy
  - Default scheduler uses ThreadPool
  - SynchronizationContextTaskScheduler for UI
- State machine tracks task status

Task Under the Hood

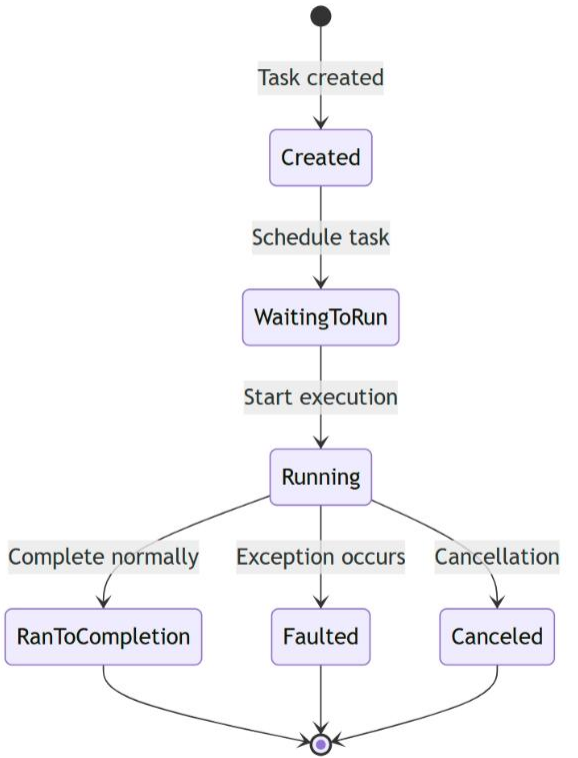
- Task is a promise/future abstraction
- Task != Thread (important distinction)
- TaskScheduler determines execution strategy
  - Default scheduler uses ThreadPool
  - SynchronizationContextTaskScheduler for UI
- State machine tracks task status
- Task continuation chains form a directed graph

Task Under the Hood

- Task is a promise/future abstraction
- Task != Thread (important distinction)
- TaskScheduler determines execution strategy
  - Default scheduler uses ThreadPool
  - SynchronizationContextTaskScheduler for UI
- State machine tracks task status
- Task continuation chains form a directed graph
- TaskCompletionSource enables manual control

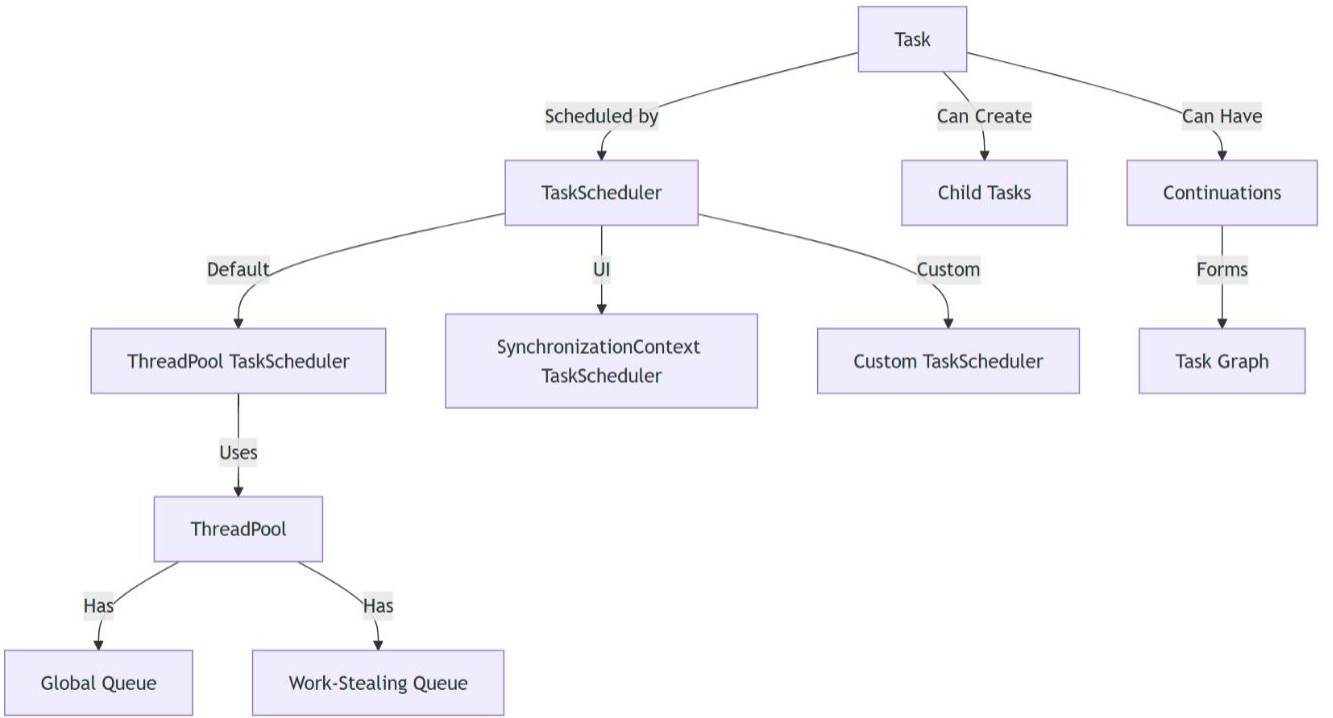
Task Under the Hood

- Task is a promise/future abstraction
- Task != Thread (important distinction)
- TaskScheduler determines execution strategy
  - Default scheduler uses ThreadPool
  - SynchronizationContextTaskScheduler for UI
- State machine tracks task status
- Task continuation chains form a directed graph
- TaskCompletionSource enables manual control



Task Scheduling and Execution

Task Scheduling and Execution



Task vs. Thread

Task vs. Thread

- A Task is a **logical** unit of work

Task vs. Thread

- A Task is a **logical** unit of work
- A Thread is a **physical** execution context

Task vs. Thread

- A Task is a **logical** unit of work
- A Thread is a **physical** execution context
- Multiple Tasks can run on a single Thread

Task vs. Thread

- A Task is a **logical** unit of work
- A Thread is a **physical** execution context
- Multiple Tasks can run on a single Thread
- Tasks can automatically use multiple Threads (via ThreadPool)

Task vs. Thread

- A Task is a **logical** unit of work
- A Thread is a **physical** execution context
- Multiple Tasks can run on a single Thread
- Tasks can automatically use multiple Threads (via ThreadPool)
- Tasks support waiting for completion, cancellation, continuations

Task Composition Patterns

Mühlehner & Tavolato GmbH

Task Composition Patterns

Sequence (Chaining)

Task Composition Patterns

Sequence (Chaining)

```
Task.Run(() ⇒ Step1())
    .ContinueWith(t ⇒ Step2(t.Result))
    .ContinueWith(t ⇒ Step3(t.Result));
```

Task Composition Patterns

Sequence (Chaining)

```
1   Task.Run(() ⇒ Step1())
2       .ContinueWith(t ⇒ Step2(t.Result))
3       .ContinueWith(t ⇒ Step3(t.Result));
```

Fan-Out (Parallel Execution)

Task Composition Patterns

Sequence (Chaining)

```
Task.Run(() ⇒ Step1())
    .ContinueWith(t ⇒ Step2(t.Result))
    .ContinueWith(t ⇒ Step3(t.Result));
```

Fan-Out (Parallel Execution)

```
Task[] tasks = new Task[3];
tasks[0] = Task.Run(() ⇒ DoWork1());
tasks[1] = Task.Run(() ⇒ DoWork2());
tasks[2] = Task.Run(() ⇒ DoWork3());
Task.WaitAll(tasks);
```

Task Composition Patterns

Sequence (Chaining)

```
Task.Run(() ⇒ Step1())
    .ContinueWith(t ⇒ Step2(t.Result))
    .ContinueWith(t ⇒ Step3(t.Result));
```

Fan-Out (Parallel Execution)

```
Task[] tasks = new Task[3];
tasks[0] = Task.Run(() ⇒ DoWork1());
tasks[1] = Task.Run(() ⇒ DoWork2());
tasks[2] = Task.Run(() ⇒ DoWork3());
Task.WaitAll(tasks);
```

Fan-In (Aggregation)

Task Composition Patterns

Sequence (Chaining)

```
Task.Run(() ⇒ Step1())
    .ContinueWith(t ⇒ Step2(t.Result))
    .ContinueWith(t ⇒ Step3(t.Result));
```

Fan-Out (Parallel Execution)

```
Task[] tasks = new Task[3];
tasks[0] = Task.Run(() ⇒ DoWork1());
tasks[1] = Task.Run(() ⇒ DoWork2());
tasks[2] = Task.Run(() ⇒ DoWork3());
Task.WaitAll(tasks);
```

Fan-In (Aggregation)

```
var task1 = Task.Run(() ⇒ GetData1());
var task2 = Task.Run(() ⇒ GetData2());
Task.WhenAll(task1, task2)
    .ContinueWith(t ⇒ {
        var results = t.Result;
        ProcessCombined(results);
    });
```

Task Composition Patterns

Sequence (Chaining)

```
Task.Run(() ⇒ Step1())
    .ContinueWith(t ⇒ Step2(t.Result))
    .ContinueWith(t ⇒ Step3(t.Result));
```

Fan-Out (Parallel Execution)

```
Task[] tasks = new Task[3];
tasks[0] = Task.Run(() ⇒ DoWork1());
tasks[1] = Task.Run(() ⇒ DoWork2());
tasks[2] = Task.Run(() ⇒ DoWork3());
Task.WaitAll(tasks);
```

Fan-In (Aggregation)

```
var task1 = Task.Run(() ⇒ GetData1());
var task2 = Task.Run(() ⇒ GetData2());
Task.WhenAll(task1, task2)
    .ContinueWith(t ⇒ {
        var results = t.Result;
        ProcessCombined(results);
    });
```

First to Complete

Task Composition Patterns

Sequence (Chaining)

```
Task.Run(() ⇒ Step1())
    .ContinueWith(t ⇒ Step2(t.Result))
    .ContinueWith(t ⇒ Step3(t.Result));
```

Fan-Out (Parallel Execution)

```
Task[] tasks = new Task[3];
tasks[0] = Task.Run(() ⇒ DoWork1());
tasks[1] = Task.Run(() ⇒ DoWork2());
tasks[2] = Task.Run(() ⇒ DoWork3());
Task.WaitAll(tasks);
```

Fan-In (Aggregation)

```
var task1 = Task.Run(() ⇒ GetData1());
var task2 = Task.Run(() ⇒ GetData2());
Task.WhenAll(task1, task2)
    .ContinueWith(t ⇒ {
        var results = t.Result;
        ProcessCombined(results);
    });
```

First to Complete

```
Task<int> t1 = Task.Run(() ⇒ Source1());
Task<int> t2 = Task.Run(() ⇒ Source2());
Task.WhenAny(t1, t2)
    .ContinueWith(t ⇒ {
        var firstResult = t.Result.Result;
        ProcessFirstResult(firstResult);
    });
```

Task Composition Patterns

Sequence (Chaining)

```
Task.Run(() ⇒ Step1())
    .ContinueWith(t ⇒ Step2(t.Result))
    .ContinueWith(t ⇒ Step3(t.Result));
```

Fan-Out (Parallel Execution)

```
Task[] tasks = new Task[3];
tasks[0] = Task.Run(() ⇒ DoWork1());
tasks[1] = Task.Run(() ⇒ DoWork2());
tasks[2] = Task.Run(() ⇒ DoWork3());
Task.WaitAll(tasks);
```

Fan-In (Aggregation)

```
var task1 = Task.Run(() ⇒ GetData1());
var task2 = Task.Run(() ⇒ GetData2());
Task.WhenAll(task1, task2)
    .ContinueWith(t ⇒ {
        var results = t.Result;
        ProcessCombined(results);
    });
```

First to Complete

```
Task<int> t1 = Task.Run(() ⇒ Source1());
Task<int> t2 = Task.Run(() ⇒ Source2());
Task.WhenAny(t1, t2)
    .ContinueWith(t ⇒ {
        var firstResult = t.Result.Result;
        ProcessFirstResult(firstResult);
    });
```

Long-Running Task

Task Composition Patterns

Sequence (Chaining)

```
Task.Run(() ⇒ Step1())
    .ContinueWith(t ⇒ Step2(t.Result))
    .ContinueWith(t ⇒ Step3(t.Result));
```

Fan-Out (Parallel Execution)

```
Task[] tasks = new Task[3];
tasks[0] = Task.Run(() ⇒ DoWork1());
tasks[1] = Task.Run(() ⇒ DoWork2());
tasks[2] = Task.Run(() ⇒ DoWork3());
Task.WaitAll(tasks);
```

Fan-In (Aggregation)

```
var task1 = Task.Run(() ⇒ GetData1());
var task2 = Task.Run(() ⇒ GetData2());
Task.WhenAll(task1, task2)
    .ContinueWith(t ⇒ {
        var results = t.Result;
        ProcessCombined(results);
    });
```

First to Complete

```
Task<int> t1 = Task.Run(() ⇒ Source1());
Task<int> t2 = Task.Run(() ⇒ Source2());
Task.WhenAny(t1, t2)
    .ContinueWith(t ⇒ {
        var firstResult = t.Result.Result;
        ProcessFirstResult(firstResult);
    });
```

Long-Running Task

```
Task.Factory.StartNew(() ⇒ {
    // Long-running, CPU-intensive work
}, TaskCreationOptions.LongRunning);
```

Task Composition Patterns

Sequence (Chaining)

```
Task.Run(() ⇒ Step1())
    .ContinueWith(t ⇒ Step2(t.Result))
    .ContinueWith(t ⇒ Step3(t.Result));
```

Fan-Out (Parallel Execution)

```
Task[] tasks = new Task[3];
tasks[0] = Task.Run(() ⇒ DoWork1());
tasks[1] = Task.Run(() ⇒ DoWork2());
tasks[2] = Task.Run(() ⇒ DoWork3());
Task.WaitAll(tasks);
```

Fan-In (Aggregation)

```
var task1 = Task.Run(() ⇒ GetData1());
var task2 = Task.Run(() ⇒ GetData2());
Task.WhenAll(task1, task2)
    .ContinueWith(t ⇒ {
        var results = t.Result;
        ProcessCombined(results);
    });
```

First to Complete

```
Task<int> t1 = Task.Run(() ⇒ Source1());
Task<int> t2 = Task.Run(() ⇒ Source2());
Task.WhenAny(t1, t2)
    .ContinueWith(t ⇒ {
        var firstResult = t.Result.Result;
        ProcessFirstResult(firstResult);
    });
```

Long-Running Task

```
Task.Factory.StartNew(() ⇒ {
    // Long-running, CPU-intensive work
}, TaskCreationOptions.LongRunning);
```

Interleaved Execution

Task Composition Patterns

Sequence (Chaining)

```
Task.Run(() ⇒ Step1())
    .ContinueWith(t ⇒ Step2(t.Result))
    .ContinueWith(t ⇒ Step3(t.Result));
```

Fan-Out (Parallel Execution)

```
Task[] tasks = new Task[3];
tasks[0] = Task.Run(() ⇒ DoWork1());
tasks[1] = Task.Run(() ⇒ DoWork2());
tasks[2] = Task.Run(() ⇒ DoWork3());
Task.WaitAll(tasks);
```

Fan-In (Aggregation)

```
var task1 = Task.Run(() ⇒ GetData1());
var task2 = Task.Run(() ⇒ GetData2());
Task.WhenAll(task1, task2)
    .ContinueWith(t ⇒ {
        var results = t.Result;
        ProcessCombined(results);
    });
```

First to Complete

```
Task<int> t1 = Task.Run(() ⇒ Source1());
Task<int> t2 = Task.Run(() ⇒ Source2());
Task.WhenAny(t1, t2)
    .ContinueWith(t ⇒ {
        var firstResult = t.Result.Result;
        ProcessFirstResult(firstResult);
    });
```

Long-Running Task

```
Task.Factory.StartNew(() ⇒ {
    // Long-running, CPU-intensive work
}, TaskCreationOptions.LongRunning);
```

Interleaved Execution

```
await Task.Yield(); // Cooperative multitasking
```

From Task to Async/Await

Mühlehner & Tavolato GmbH

From Task to Async/Await

Without Async/Await

Mühlehner & Tavolato GmbH

From Task to Async/Await

Without Async/Await

```
Task<string> DownloadAsync(string url)
{
    var tcs = new TaskCompletionSource<string>();

    WebClient client = new WebClient();
    client.DownloadStringCompleted += (s, e) => {
        if (e.Error != null)
            tcs.SetException(e.Error);
        else if (e.Cancelled)
            tcs.SetCanceled();
        else
            tcs.SetResult(e.Result);
    };

    client.DownloadStringAsync(new Uri(url));

    return tcs.Task;
}

// Usage
DownloadAsync("http://example.com")
    .ContinueWith(t => {
        try {
            Console.WriteLine(t.Result);
        } catch (AggregateException ex) {
            Console.WriteLine(ex.InnerException);
        }
    });
```

From Task to Async/Await

Without Async/Await

```
Task<string> DownloadAsync(string url)
{
    var tcs = new TaskCompletionSource<string>();

    WebClient client = new WebClient();
    client.DownloadStringCompleted += (s, e) => {
        if (e.Error ≠ null)
            tcs.SetException(e.Error);
        else if (e.Cancelled)
            tcs.SetCanceled();
        else
            tcs.SetResult(e.Result);
    };

    client.DownloadStringAsync(new Uri(url));

    return tcs.Task;
}

// Usage
DownloadAsync("http://example.com")
    .ContinueWith(t ⇒ {
        try {
            Console.WriteLine(t.Result);
        } catch (AggregateException ex) {
            Console.WriteLine(ex.InnerException);
        }
    });
```

With Async/Await

From Task to Async/Await

Without Async/Await

```
Task<string> DownloadAsync(string url)
{
    var tcs = new TaskCompletionSource<string>();

    WebClient client = new WebClient();
    client.DownloadStringCompleted += (s, e) => {
        if (e.Error ≠ null)
            tcs.SetException(e.Error);
        else if (e.Cancelled)
            tcs.SetCanceled();
        else
            tcs.SetResult(e.Result);
    };

    client.DownloadStringAsync(new Uri(url));

    return tcs.Task;
}

// Usage
DownloadAsync("http://example.com")
    .ContinueWith(t ⇒ {
        try {
            Console.WriteLine(t.Result);
        } catch (AggregateException ex) {
            Console.WriteLine(ex.InnerException);
        }
    });
```

With Async/Await

```
async Task<string> DownloadAsync(string url)
{
    using (WebClient client = new WebClient())
    {
        return await client.DownloadStringTaskAsync(url);
    }
}

// Usage
try
{
    string result = await DownloadAsync("http://example.com");
    Console.WriteLine(result);
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}
```

7. Async/Await Revolution

```
async Task<string> GetWebContentAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("Starting download...");

        // Asynchronously get string without blocking
        string result = await client.GetStringAsync(url);

        Console.WriteLine("Download completed!");
        return result.Substring(0, 100);
    }
}

// Usage
string content = await GetWebContentAsync("http://example.com");
Console.WriteLine(content);
```

7. Async/Await Revolution

- Introduced in C# 5.0 (.NET 4.5)

```
async Task<string> GetWebContentAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("Starting download...");

        // Asynchronously get string without blocking
        string result = await client.GetStringAsync(url);

        Console.WriteLine("Download completed!");
        return result.Substring(0, 100);
    }
}

// Usage
string content = await GetWebContentAsync("http://example.com");
Console.WriteLine(content);
```

7. Async/Await Revolution

- Introduced in C# 5.0 (.NET 4.5)
- Syntactic sugar over task-based pattern

```
async Task<string> GetWebContentAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("Starting download...");

        // Asynchronously get string without blocking
        string result = await client.GetStringAsync(url);

        Console.WriteLine("Download completed!");
        return result.Substring(0, 100);
    }
}

// Usage
string content = await GetWebContentAsync("http://example.com");
Console.WriteLine(content);
```

7. Async/Await Revolution

- Introduced in C# 5.0 (.NET 4.5)
- Syntactic sugar over task-based pattern
- Transforms synchronous-looking code into state machines

```
async Task<string> GetWebContentAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("Starting download...");

        // Asynchronously get string without blocking
        string result = await client.GetStringAsync(url);

        Console.WriteLine("Download completed!");
        return result.Substring(0, 100);
    }
}

// Usage
string content = await GetWebContentAsync("http://example.com");
Console.WriteLine(content);
```

7. Async/Await Revolution

- Introduced in C# 5.0 (.NET 4.5)
- Syntactic sugar over task-based pattern
- Transforms synchronous-looking code into state machines
- Compiler-generated continuations

```
async Task<string> GetWebContentAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("Starting download...");

        // Asynchronously get string without blocking
        string result = await client.GetStringAsync(url);

        Console.WriteLine("Download completed!");
        return result.Substring(0, 100);
    }
}

// Usage
string content = await GetWebContentAsync("http://example.com");
Console.WriteLine(content);
```

7. Async/Await Revolution

- Introduced in C# 5.0 (.NET 4.5)
- Syntactic sugar over task-based pattern
- Transforms synchronous-looking code into state machines
- Compiler-generated continuations
- Automatic SynchronizationContext capture and restore

```
async Task<string> GetWebContentAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("Starting download...");

        // Asynchronously get string without blocking
        string result = await client.GetStringAsync(url);

        Console.WriteLine("Download completed!");
        return result.Substring(0, 100);
    }
}

// Usage
string content = await GetWebContentAsync("http://example.com");
Console.WriteLine(content);
```

7. Async/Await Revolution

- Introduced in C# 5.0 (.NET 4.5)
- Syntactic sugar over task-based pattern
- Transforms synchronous-looking code into state machines
- Compiler-generated continuations
- Automatic SynchronizationContext capture and restore
- Natural error handling with try/catch

```
async Task<string> GetWebContentAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("Starting download...");

        // Asynchronously get string without blocking
        string result = await client.GetStringAsync(url);

        Console.WriteLine("Download completed!");
        return result.Substring(0, 100);
    }
}

// Usage
string content = await GetWebContentAsync("http://example.com");
Console.WriteLine(content);
```

7. Async/Await Revolution

- Introduced in C# 5.0 (.NET 4.5)
- Syntactic sugar over task-based pattern
- Transforms synchronous-looking code into state machines
- Compiler-generated continuations
- Automatic SynchronizationContext capture and restore
- Natural error handling with try/catch
- Dramatically simplified asynchronous code

```
async Task<string> GetWebContentAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        Console.WriteLine("Starting download...");

        // Asynchronously get string without blocking
        string result = await client.GetStringAsync(url);

        Console.WriteLine("Download completed!");
        return result.Substring(0, 100);
    }
}

// Usage
string content = await GetWebContentAsync("http://example.com");
Console.WriteLine(content);
```

Async/Await Under the Hood: The State Machine

Async/Await Under the Hood: The State Machine

A Simplified Example

```
1   // Your async method
2   async Task<int> ExampleAsync()
3   {
4       Console.WriteLine("Starting");
5       int value = await Task.FromResult(42);
6       Console.WriteLine("Middle");
7       int secondValue = await Task.FromResult(value + 1);
8       Console.WriteLine("End");
9       return secondValue;
10  }
```

What the Compiler Generates (Conceptually)

```
Task<int> ExampleAsync()
{
    // Create state machine instance
    var stateMachine = new ExampleAsyncStateMachine();
    stateMachine.builder = AsyncTaskMethodBuilder<int>.Create();
    stateMachine.state = -1; // Initial state
    stateMachine.this = this; // Capture this pointer

    // Start the state machine
    stateMachine.builder.Start(ref stateMachine);
    return stateMachine.builder.Task;
}
```

Async State Machine Implementation

```
1   struct ExampleAsyncStateMachine : IAsyncStateMachine
2   {
3       // Fields to store local variables and parameters
4       public int state;
5       public AsyncTaskMethodBuilder<int> builder;
6       public int value;
7       public int secondValue;
8       public Task<int> awaiter1;
9       public Task<int> awaiter2;
```

```
    // Method that implements the state machine logic
    void IAsyncStateMachine.MoveNext()
    {
        int result;
        try
        {
            bool completed = false;
            switch (state)
            {
                case -1: // Initial state
                    Console.WriteLine("Starting");
                    awaiter1 = Task.FromResult(42);
                    if (awaiter1.IsCompleted)
                    {
                        goto case 0; // Fast path if already completed
                    }
                    state = 0; // Next state after awaiting
                    builder.AwaitUnsafeOnCompleted(ref awaiter1, ref this);
                    return;

                case 0: // After first await
                    value = awaiter1.Result;
                    awaiter1 = null; // Release reference
                    Console.WriteLine("Middle");
                    awaiter2 = Task.FromResult(value + 1);
                    if (awaiter2.IsCompleted)
                    {
                        goto case 1; // Fast path if already completed
                    }
                    state = 1; // Next state after awaiting
                    builder.AwaitUnsafeOnCompleted(ref awaiter2, ref this);
                    return;

                case 1: // After second await
                    secondValue = awaiter2.Result;
                    awaiter2 = null; // Release reference
                    Console.WriteLine("End");
                    result = secondValue;
                    completed = true;
                    break;
            }

            if (completed)
            {
                // Set the result and mark as complete
                builder.SetResult(result);
            }
        }
```

Continuation and Context Capture

Mühlehner & Tavolato GmbH

Continuation and Context Capture

SynchronizationContext Capture

Mühlehner & Tavolato GmbH

Continuation and Context Capture

SynchronizationContext Capture

```
// In UI thread
private async void Button_Click(object sender, EventA
{
    // Captures UI SynchronizationContext
    await DoWorkAsync();

    // Back on UI thread!
    UpdateUI(); // Safe to call
}
```

Continuation and Context Capture

SynchronizationContext Capture

```
1   // In UI thread
2   private async void Button_Click(object sender, EventA
3   {
4       // Captures UI SynchronizationContext
5       await DoWorkAsync();
6
7       // Back on UI thread!
8       UpdateUI(); // Safe to call
9   }
```

How It Works

Continuation and Context Capture

SynchronizationContext Capture

```
1   // In UI thread
2   private async void Button_Click(object sender, EventA
3   {
4       // Captures UI SynchronizationContext
5       await DoWorkAsync();
6
7       // Back on UI thread!
8       UpdateUI(); // Safe to call
9   }
```

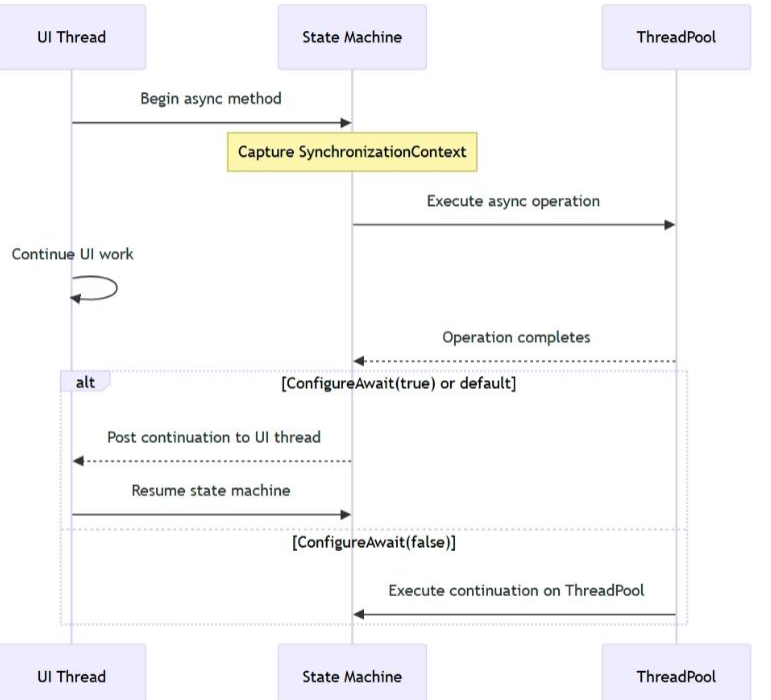
How It Works

- Current SynchronizationContext is captured before await
- Continuations are posted back to that context
- Makes UI programming much simpler
- ConfigureAwait(false) can disable this

Continuation and Context Capture

SynchronizationContext Capture

```
1   // In UI thread
2   private async void Button_Click(object sender, EventA
3   {
4       // Captures UI SynchronizationContext
5       await DoWorkAsync();
6
7       // Back on UI thread!
8       UpdateUI(); // Safe to call
9   }
```



How It Works

- Current SynchronizationContext is captured before await
- Continuations are posted back to that context
- Makes UI programming much simpler
- ConfigureAwait(false) can disable this

Await Mechanics: ConfigureAwait

Mühlehner & Tavolato GmbH

Await Mechanics: ConfigureAwait

Default Behavior

Mühlehner & Tavolato GmbH

Await Mechanics: ConfigureAwait

Default Behavior

```
// Captures and returns to current context
await someTask;

// Equivalent to:
await someTask.ConfigureAwait(true);
```

Await Mechanics: ConfigureAwait

Default Behavior

```
// Captures and returns to current context
await someTask;

// Equivalent to:
await someTask.ConfigureAwait(true);
```

Avoiding Context Capture

Await Mechanics: ConfigureAwait

Default Behavior

```
// Captures and returns to current context
await someTask;

// Equivalent to:
await someTask.ConfigureAwait(true);
```

Avoiding Context Capture

```
// Doesn't capture or return to original context
await someTask.ConfigureAwait(false);
```

Await Mechanics: ConfigureAwait

Default Behavior

```
// Captures and returns to current context
await someTask;

// Equivalent to:
await someTask.ConfigureAwait(true);
```

Avoiding Context Capture

```
// Doesn't capture or return to original context
await someTask.ConfigureAwait(false);
```

Best Practices

Await Mechanics: ConfigureAwait

Default Behavior

```
// Captures and returns to current context
await someTask;

// Equivalent to:
await someTask.ConfigureAwait(true);
```

Avoiding Context Capture

```
// Doesn't capture or return to original context
await someTask.ConfigureAwait(false);
```

Best Practices

- Use `ConfigureAwait(false)` in library code
- Keep default in UI applications
- Library and UI code have different needs
- Avoid deadlocks with `ConfigureAwait(false)`

Performance Implications

```
// UI app - keep default for UI safety
private async void Button_Click(object sender, EventArgs e)
{
    var result = await GetDataAsync();
    resultTextBox.Text = result; // Needs UI context
}

// Library code - avoid unneeded context
public async Task<string> GetDataAsync()
{
    // No UI dependency, skip context capture
    var data = await httpClient.GetStringAsync(url)
        .ConfigureAwait(false);

    // CPU-bound work doesn't need UI thread
    return Process(data);
}
```

Async Method Return Types

Async Method Return Types

Task

Async Method Return Types

Task

```
async Task DoWorkAsync()
{
    await Task.Delay(1000);
    // No return value needed
}
```

Async Method Return Types

Task

```
async Task DoWorkAsync()
{
    await Task.Delay(1000);
    // No return value needed
}
```

`Task<T>`

Async Method Return Types

Task

```
async Task DoWorkAsync()
{
    await Task.Delay(1000);
    // No return value needed
}
```

Task<T>

```
async Task<int> GetValueAsync()
{
    await Task.Delay(1000);
    return 42; // Returns a value
}
```

ValueTask / `ValueTask<T>` (.NET Core 2.0+)

ValueTask / `ValueTask<T>` (.NET Core 2.0+)

```
async ValueTask<int> GetCachedValueAsync(int key)
{
    if (_cache.TryGetValue(key, out int value))
    {
        return value; // No Task allocation needed
    }

    value = await ComputeValueAsync(key);
    _cache[key] = value;
    return value;
}
```

ValueTask / `ValueTask<T>` (.NET Core 2.0+)

```
async ValueTask<int> GetCachedValueAsync(int key)
{
    if (_cache.TryGetValue(key, out int value))
    {
        return value; // No Task allocation needed
    }

    value = await ComputeValueAsync(key);
    _cache[key] = value;
    return value;
}
```

Synchronous Completion

ValueTask / `ValueTask<T>` (.NET Core 2.0+)

```
async ValueTask<int> GetCachedValueAsync(int key)
{
    if (_cache.TryGetValue(key, out int value))
    {
        return value; // No Task allocation needed
    }

    value = await ComputeValueAsync(key);
    _cache[key] = value;
    return value;
}
```

Synchronous Completion

- Avoids allocating Task when result is available

ValueTask / `ValueTask<T>` (.NET Core 2.0+)

```
async ValueTask<int> GetCachedValueAsync(int key)
{
    if (_cache.TryGetValue(key, out int value))
    {
        return value; // No Task allocation needed
    }

    value = await ComputeValueAsync(key);
    _cache[key] = value;
    return value;
}
```

Synchronous Completion

- Avoids allocating Task when result is available
- Better performance for cache hits/fast paths

ValueTask / `ValueTask<T>` (.NET Core 2.0+)

```
async ValueTask<int> GetCachedValueAsync(int key)
{
    if (_cache.TryGetValue(key, out int value))
    {
        return value; // No Task allocation needed
    }

    value = await ComputeValueAsync(key);
    _cache[key] = value;
    return value;
}
```

Synchronous Completion

- Avoids allocating Task when result is available
- Better performance for cache hits/fast paths
- Small struct instead of reference type

ValueTask / `ValueTask<T>` (.NET Core 2.0+)

```
async ValueTask<int> GetCachedValueAsync(int key)
{
    if (_cache.TryGetValue(key, out int value))
    {
        return value; // No Task allocation needed
    }

    value = await ComputeValueAsync(key);
    _cache[key] = value;
    return value;
}
```

Synchronous Completion

- Avoids allocating Task when result is available
- Better performance for cache hits/fast paths
- Small struct instead of reference type
- Designed for high-performance scenarios

8. Modern Async Features in C# (C# 8.0+)

Async Streams (IAsyncEnumerable)

```
async IAsyncEnumerable<int> GenerateSequenceAsync()
{
    for (int i = 0; i < 10; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}

// Consuming
await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}
```

Async Disposable (IAsyncDisposable)

```
await using (var resource = new AsyncResource())
{
    await resource.UseAsync();
} // Automatically calls DisposeAsync
```

Parallel Foreach with ForEachAsync

```
// Process multiple items in parallel with throttling
await Parallel.ForEachAsync(
    items,
    new ParallelOptions { MaxDegreeOfParallelism = 10 },
    async (item, token) =>
    {
        await ProcessItemAsync(item, token);
    });
```

Default Interface Methods with Async

```
interface IDataProcessor
{
    // Default async implementation in interface
    async Task<int> ProcessAsync(Data data)
    {
        // Default processing logic
        await Task.Delay(100);
        return data.Value;
    }
}
```

Async Streams Under the Hood

`IAsyncEnumerable<T>`

```
1   public interface IAsyncEnumerable<out T>
2   {
3       IAsyncEnumerator<T> GetAsyncEnumerator(
4           CancellationToken cancellationToken = default);
5   }
6
7   public interface IAsyncEnumerator<out T> : IAsyncDisposable
8   {
9       ValueTask<bool> MoveNextAsync();
10      T Current { get; }
11  }
12
13  public interface IAsyncDisposable
14  {
15      ValueTask DisposeAsync();
16  }
```

Async Streams Under the Hood

`IAsyncEnumerable<T>`

```
1  public interface IAsyncEnumerable<out T>
2  {
3      IAsyncEnumerator<T> GetAsyncEnumerator(
4          CancellationToken cancellationToken = default);
5  }
6
7  public interface IAsyncEnumerator<out T> : IAsyncDisposable
8  {
9      ValueTask<bool> MoveNextAsync();
10     T Current { get; }
11 }
12
13 public interface IAsyncDisposable
14 {
15     ValueTask DisposeAsync();
16 }
```

State Machine Transformation

- Similar to regular async methods
- But combines with iterator state machine
- Created as nested type in compiler-generated code
- Handles both async and enumeration states
- Uses ValueTask for efficiency
- Built-in cancellation support

Cancellation in Async Code

Mühlehner & Tavolato GmbH

Cancellation in Async Code

Creating Cancellation Tokens

Mühlehner & Tavolato GmbH

Cancellation in Async Code

Creating Cancellation Tokens

```
// Create a token source with timeout
using var cts = new CancellationTokenSource(
    TimeSpan.FromSeconds(5));

// Manual cancellation
var cts = new CancellationTokenSource();
cts.Cancel();

// Linked tokens
using var linkedCts = CancellationTokenSource
    .CreateLinkedTokenSource(token1, token2);
```

Cancellation in Async Code

Creating Cancellation Tokens

```
// Create a token source with timeout
using var cts = new CancellationTokenSource(
    TimeSpan.FromSeconds(5));

// Manual cancellation
var cts = new CancellationTokenSource();
cts.Cancel();

// Linked tokens
using var linkedCts = CancellationTokenSource
    .CreateLinkedTokenSource(token1, token2);
```

Passing Tokens

Cancellation in Async Code

Creating Cancellation Tokens

```
// Create a token source with timeout
using var cts = new CancellationTokenSource(
    TimeSpan.FromSeconds(5));

// Manual cancellation
var cts = new CancellationTokenSource();
cts.Cancel();

// Linked tokens
using var linkedCts = CancellationTokenSource
    .CreateLinkedTokenSource(token1, token2);
```

Passing Tokens

```
async Task<string> GetDataAsync(
    string url,
    CancellationToken token = default)
{
    return await httpClient.GetStringAsync(url, token);
}
```

Checking for Cancellation

Checking for Cancellation

```
async Task LongRunningAsync(CancellationToken token)
{
    for (int i = 0; i < 1000; i++)
    {
        // Check periodically
        token.ThrowIfCancellationRequested();

        await DoWorkAsync();
    }
}
```

Checking for Cancellation

```
async Task LongRunningAsync(CancellationToken token)
{
    for (int i = 0; i < 1000; i++)
    {
        // Check periodically
        token.ThrowIfCancellationRequested();

        await DoWorkAsync();
    }
}
```

Creating Cancellable Tasks

Checking for Cancellation

```
async Task LongRunningAsync(CancellationToken token)
{
    for (int i = 0; i < 1000; i++)
    {
        // Check periodically
        token.ThrowIfCancellationRequested();

        await DoWorkAsync();
    }
}
```

Creating Cancellable Tasks

```
Task task = Task.Delay(1000, token);

// Timeout pattern
Task<T> taskWithTimeout = Task.WhenAny(
    actualTask,
    Task.Delay(timeout, token))
    .ContinueWith(t =>
    {
        if (t.Result != actualTask)
            throw new TimeoutException();
        return actualTask.Result;
    });
```

Progress Reporting Pattern

Mühlehner & Tavolato GmbH

Progress Reporting Pattern

`IProgress<T>` Interface

Progress Reporting Pattern

`IProgress<T>` Interface

```
1   public interface IProgress<in T>
2   {
3       void Report(T value);
4   }
```

Progress Reporting Pattern

`IProgress<T>` Interface

```
1   public interface IProgress<in T>
2   {
3       void Report(T value);
4   }
```

Creating Progress Handler

Progress Reporting Pattern

`IProgress<T>` Interface

```
1   public interface IProgress<in T>
2   {
3       void Report(T value);
4   }
```

Creating Progress Handler

```
1   var progress = new Progress<int>(percent ⇒
2   {
3       progressBar.Value = percent;
4       statusLabel.Text = $"{percent}% complete";
5   });
```

Using Progress in Async Methods

Using Progress in Async Methods

```
async Task DownloadFileAsync(
    string url,
    string destination,
    IProgress<int> progress = null,
    CancellationToken token = default)
{
    using var httpClient = new HttpClient();
    using var response = await httpClient.GetAsync(
        url,
        HttpCompletionOption.ResponseHeadersRead,
        token);

    var totalBytes = response.Content.Headers.ContentLength ?? -1L;
    var bytesRead = 0L;

    using var contentStream = await response.Content.ReadAsStreamAsync();
    using var fileStream = new FileStream(destination, FileMode.Create);

    var buffer = new byte[8192];
    int read;

    while ((read = await contentStream.ReadAsync(buffer, 0, buffer.Length, token)) > 0)
    {
        await fileStream.WriteAsync(buffer, 0, read, token);

        bytesRead += read;
        if (totalBytes > 0 && progress != null)
        {
            var progressPercentage = (int)((bytesRead * 100) / totalBytes);
            progress.Report(progressPercentage);
        }
    }
}
```

Error Handling in Async/Await

```
Task task = Task.Run(() => { throw new Exception("Error"); });

try
{
    // Throws AggregateException containing the original
    task.Wait();
}
catch (AggregateException ae)
{
    foreach (var ex in ae.InnerExceptions)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Unobserved Task Exceptions

```
1   // Dangerous! Exception will be unobserved
2   Task.Run(() ⇒ { throw new Exception("Error"); });
3
4   // Safe: observe the exception
5   Task.Run(() ⇒ { throw new Exception("Error"); })
6       .ContinueWith(t ⇒
7       {
8           if (t.IsFaulted)
9               Console.WriteLine(t.Exception);
10      });
```

Async/Await Error Handling

```
async Task DoWorkAsync()
{
    try
    {
        await Task.Run(() => { throw new Exception("Error"); });
    }
    catch (Exception ex) // Unwrapped automatically!
    {
        Console.WriteLine(ex.Message);
    }
}
```

Multiple Errors

```
async Task MultipleErrorsAsync()
{
    // Create multiple tasks that may fail
    var tasks = new List<Task>();
    for (int i = 0; i < 10; i++)
    {
        int taskNum = i;
        tasks.Add(Task.Run(() =>
        {
            if (taskNum % 3 == 0)
                throw new Exception($"Task {taskNum} failed");
        }));
    }

    try
    {
        // WhenAll preserves all exceptions
        await Task.WhenAll(tasks);
    }
    catch (Exception)
    {
        // Find all the failed tasks
        foreach (var task in tasks)
        {
            if (task.IsFaulted)
            {
                Console.WriteLine(task.Exception.InnerException.Message);
            }
        }
    }
}
```

Advanced Async Patterns

Lazy Initialization

Mühlehner & Tavolato GmbH

Advanced Async Patterns

Lazy Initialization

```
1   private AsyncLazy<DbConnection> _connection;
2
3   public class AsyncLazy<T> : Lazy<Task<T>>
4   {
5       public AsyncLazy(Func<Task<T>> valueFactory) :
6           base(valueFactory) { }
7   }
8
9   // Usage
10  _connection = new AsyncLazy<DbConnection>(async () ⇒
11  {
12      var conn = new SqlConnection(connectionString);
13      await conn.OpenAsync();
14      return conn;
15  });
16
17  // Later
18  DbConnection conn = await _connection.Value;
```

Interleaved Operations

```
async Task InterleavedOperationsAsync()
{
    // Start multiple operations
    Task<int> task1 = SlowOperationAsync(1);
    Task<int> task2 = SlowOperationAsync(2);
    Task<int> task3 = SlowOperationAsync(3);

    // Process results as they arrive
    while (new[] { task1, task2, task3 }.Any(t => !t.IsCompleted))
    {
        Task<int> completed = await Task.WhenAny(
            task1, task2, task3);

        if (completed == task1)
        {
            Console.WriteLine($"Task 1 completed: {await task1}");
            task1 = Task.FromResult(-1); // Mark as processed
        }
        else if (completed == task2)
        {
            Console.WriteLine($"Task 2 completed: {await task2}");
            task2 = Task.FromResult(-1); // Mark as processed
        }
        else if (completed == task3)
        {
            Console.WriteLine($"Task 3 completed: {await task3}");
            task3 = Task.FromResult(-1); // Mark as processed
        }
    }
}
```

Synchronization vs. Asynchrony

Synchronization vs. Asynchrony

Asynchrony is Not About Concurrency

- Asynchrony is about efficient resource usage
- Tasks can run sequentially, not just concurrently
- Main benefit: thread utilization during I/O waits
- Synchronization is still needed for shared state
- Async code can still have race conditions

Async/Await Best Practices

Mühlehner & Tavolato GmbH

Async/Await Best Practices

Do’s

Async/Await Best Practices

Do’s

- Use async/await all the way through the call stack
- Always pass CancellationToken parameters
- Use ConfigureAwait(false) in libraries
- Prefer Task-returning methods over void
- Use Task.WhenAll/WhenAny for composition
- Handle exceptions with standard try/catch
- Name async methods with Async suffix

Async/Await Best Practices

Do’s

- Use async/await all the way through the call stack
- Always pass CancellationToken parameters
- Use ConfigureAwait(false) in libraries
- Prefer Task-returning methods over void
- Use Task.WhenAll/WhenAny for composition
- Handle exceptions with standard try/catch
- Name async methods with Async suffix

Don’ts

Async/Await Best Practices

Do's

- Use async/await all the way through the call stack
- Always pass CancellationToken parameters
- Use ConfigureAwait(false) in libraries
- Prefer Task-returning methods over void
- Use Task.WhenAll/WhenAny for composition
- Handle exceptions with standard try/catch
- Name async methods with Async suffix

Don'ts

- Don't use async void except for event handlers
- Don't block on async code with .Result or .Wait()
- Don't use Task.Run for I/O operations
- Don't use async just to return a completed Task
- Don't capture UI context when not needed
- Don't forget to dispose of IAsyncDisposable
- Don't ignore Task returns (fire and forget)

Common Async/Await Pitfalls

Deadlocks

```
// Deadlock pattern - DON'T DO THIS
void Button_Click(object sender, EventArgs e)
{
    // Blocks UI thread while waiting for task
    // that needs UI thread to complete
    var result = GetDataAsync().Result;
}

async Task<string> GetDataAsync()
{
    await Task.Delay(1000); // Captures UI context
    return "Data";
}
```

Unobserved Exceptions

```
// Fire and forget - DON'T DO THIS
async void Button_Click(object sender, EventArgs e)
{
    // Exception will crash the app
    await RiskyOperationAsync();
}

// BETTER:
async void Button_Click(object sender, EventArgs e)
{
    try
    {
        await RiskyOperationAsync();
    }
    catch (Exception ex)
    {
        // Handle exception
    }
}
```

Async Performance Considerations

Mühlehner & Tavolato GmbH

Async Performance Considerations

Memory Overhead

Mühlehner & Tavolato GmbH

Async Performance Considerations

Memory Overhead

- State machines allocate memory
- Each continuation is an allocation
- Captured variables (closures) allocate
- Task objects themselves allocate heap memory

Async Performance Considerations

Memory Overhead

- State machines allocate memory
- Each continuation is an allocation
- Captured variables (closures) allocate
- Task objects themselves allocate heap memory

Optimization Techniques

Async Performance Considerations

Memory Overhead

- State machines allocate memory
- Each continuation is an allocation
- Captured variables (closures) allocate
- Task objects themselves allocate heap memory

Optimization Techniques

- Use ValueTask for fast-path returns
- Pool buffers and other resources
- Consider synchronous alternatives for CPU-bound work
- Use `ArrayPool<T>` for buffer management
- Consider `Span<T>` and `Memory<T>` for zero-copy IO

Value Types for Performance

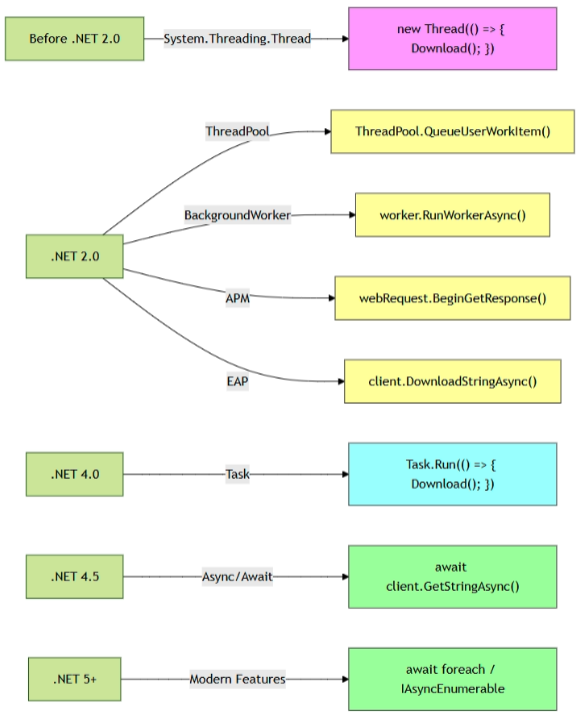
```
1   // Efficient and avoids allocations when cached
2   async ValueTask<int> GetValueAsync(int key)
3   {
4       if (_cache.TryGetValue(key, out int value))
5       {
6           return value; // No allocation!
7       }
8
9       value = await ExpensiveOperationAsync(key);
10      _cache[key] = value;
11      return value;
12  }
```

Memory Efficient IO

```
async Task ProcessFileAsync(string path)
{
    using var file = new FileStream(path, FileMode.Open);

    // Rent a buffer instead of allocating a new one
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);
    try
    {
        int bytesRead;
        while ((bytesRead = await file.ReadAsync(
            buffer, 0, buffer.Length)) > 0)
        {
            // Process the data in the buffer
            await ProcessDataAsync(
                buffer.AsMemory(0, bytesRead));
        }
    }
    finally
    {
        // Return the buffer to the pool
        ArrayPool<byte>.Shared.Return(buffer);
    }
}
```

Comparison: Evolution of a Download Operation

Comparison: Evolution of a Download Operation



Mühlehner & Tavolato GmbH

Embracing asynchronous programming with C# empowers you to build responsive, scalable, and efficient applications.

Documentation · GitHub

Mühlehner & Tavolato GmbH