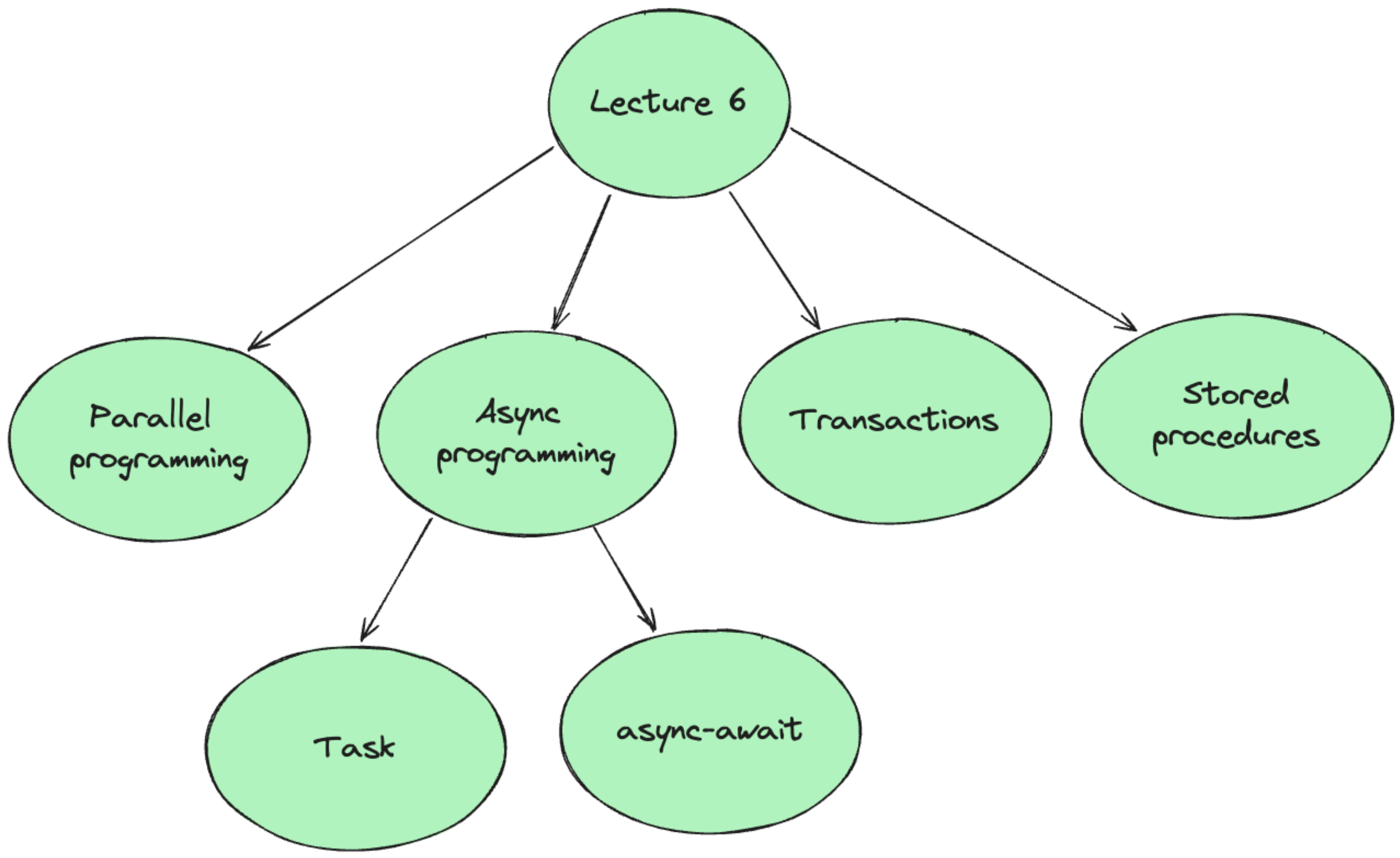


Parallel programming in C#

Parallel programming involves executing multiple tasks or computations simultaneously, leveraging multi-core processors to improve performance. In C#, the `System.Threading.Tasks` namespace provides tools to create and manage tasks that can run in parallel.

Parallel programming can be implemented using both processes and threads, which are two fundamental units of execution in operating systems.

Process

A **process** is an instance of a program running in its own isolated memory space. Each process has a complete set of its own variables, system resources, and memory. This isolation means that processes do not normally share memory or resources with other processes.

Parallel Programming with Processes

Parallel programming with processes involves running multiple processes concurrently. This is typically managed using inter-process communication (IPC) mechanisms such as sockets, shared files, message passing, or shared memory (where supported). This approach is generally more resource-intensive due to the overhead of process creation and IPC.

Advantages:

- **Isolation:** Faults in one process do not affect others.
- **Security:** Processes can be sandboxed more effectively than threads.

Disadvantages:

- **Overhead:** Processes require more time and system resources to create and manage.
- **Complexity:** Inter-process communication is more complex and slower than inter-thread communication.

Threads

A **thread** is the smallest unit of processing that can be scheduled by an operating system. It is a component of a process, and multiple threads can exist within the same process, sharing the process's memory and resources.

Parallel Programming with Threads

Using threads allows a program to perform concurrent operations within a single process. This is less resource-intensive than using multiple processes because threads share the process's memory space and resources, making communication between them faster and more straightforward.

Advantages:

- **Efficiency:** Threads are lighter than processes in terms of system resource requirements.
- **Speed:** Communication between threads is typically faster than between processes due to shared memory.

Disadvantages:

- **Safety:** Threads share the same memory space, which can lead to issues like race conditions if not managed correctly.
- **Complexity in Debugging:** Debugging multithreaded applications can be more complex due to synchronization issues and timing.

Choosing Between Processes and Threads

The choice between using processes or threads for parallel programming depends on several factors:

- **Safety and Isolation:** If isolation from faults and security is a priority, processes might be preferred despite the overhead.
- **Resource Utilization and Overhead:** If the overhead of creating and managing separate processes is too high, or if the application benefits from tight integration and fast communication, threads might be better.
- **Development Complexity:** Threads can be easier to implement within the same application but require careful handling to avoid issues with shared state.

Practical Examples

Processes Example:
A web server handling multiple requests by spawning a new process for each request ensures that a fault in processing one request does not crash the entire server.

Threads Example:
A video game might use multiple threads within the same process: one for rendering graphics, another for handling user input, and yet another for game logic, all sharing access to the game's state.

CPU-bound operations and I/O bound operations

CPU-bound operations are those that spend most of their time utilizing the Central Processing Unit (CPU) to perform computations. In other words, the speed and performance of these operations are primarily limited by the speed of the CPU. These tasks involve significant calculation that requires processing power.

I/O-bound operations are those that spend most of their time waiting for external operations to complete rather than using the CPU. These external operations could involve reading from or writing to a disk, making network requests, or interacting with databases. The performance of I/O-bound operations is limited by the speed of the I/O subsystem rather than the processor speed.

Key Differences in Concurrency Handling

- **CPU-bound** tasks benefit more from multiprocessing because this approach can spread the tasks across multiple processors or cores, thus reducing the time taken to complete the operations by performing them in parallel.
- **I/O-bound** tasks benefit more from multithreading or asynchronous programming. These methods allow a single application to handle multiple I/O operations in the background.
- While one thread is waiting for its I/O operation to complete, another thread can process other tasks. In modern programming, asynchronous programming models (like async/await in C# and Python) are particularly effective because they make it easier to write non-blocking code that can handle many I/O tasks concurrently without complex threading logic.

Manual way to deal with threads

Example 1:

```
public static void Main(string[] args)
{
    // Creating and starting the first thread
    Thread thread1 = new Thread(PrintNumbers);
    thread1.Start();

    // Creating and starting the second thread
    Thread thread2 = new Thread(PrintNumbers);
    thread2.Start();
    // Wait for both threads to complete
    thread1.Join();
    thread2.Join();

    Console.WriteLine("Both threads have completed their
execution.");
}

static void PrintNumbers()
{
    for (int i = 1; i <= 5; i++)
    {        Console.WriteLine($"Thread
{Thread.CurrentThread.ManagedThreadId}: {i}");
        Thread.Sleep(1000);  // Simulating work
    }
}
```

Example 2 - using ThreadPool:

```
public static void Main(string[] args)
{
    // Queue the task to run on the ThreadPool
    ThreadPool.QueueUserWorkItem(_ => PrintNumbers());
    ThreadPool.QueueUserWorkItem(_ => PrintNumbers());

    // Wait a little for the threads to complete (in a real
application, better synchronization should be used)
    Thread.Sleep(2000);

    Console.WriteLine("Both threads have completed their
execution.");
}

static void PrintNumbers()
{
    for (int i = 1; i <= 5; i++)
    {        Console.WriteLine($"Thread
{Thread.CurrentThread.ManagedThreadId}: {i}");
        Thread.Sleep(1000);  // Simulating work
    }
}
```

Asynchronous programming in C#

Asynchronous programming is about performing long-running tasks without blocking the main thread, keeping the application responsive. In C#, async programming is often used for I/O-bound tasks, such as reading files or querying a database.

Async programming in web applications

Asynchronous programming is particularly important for web applications due to its impact on scalability, responsiveness, and overall user experience.

Consider a web application like an online store, which must handle user requests to view products, place orders, and check out. If these processes are handled synchronously, the server might become a bottleneck under heavy load, leading to slow response times and a poor user experience. By implementing asynchronous processing for database access and external services (like payment processing), the server can handle more requests concurrently, leading to faster response times and a smoother user experience.

How async programming in handled

In C# there are various ways in which we can handle an async code.

Asynchronous programming model (APM)

The Asynchronous Programming Model (APM), also known as the IAsyncResult pattern, was one of the earliest ways to handle asynchronous operations in .NET. It involves calling methods that start with `Begin` and end with `End`.

- APM is the oldest model, which uses `BeginMethod` and `EndMethod` naming conventions. Operations start with `Begin` and end with `End`, and they usually return an `IAsyncResult`.

- Example methods include `FileStream.BeginRead` and `FileStream.EndRead`

When to Use:

- Use APM when dealing with legacy code or libraries that only support this model.
- APM is less common in new development because it's more complex and less readable compared to newer models.

Example:

```
using System;
using System.IO;
using System.Net;

public class ApmExample
{
    public static void Main()
    {
        string url = "http://example.com";
        WebClient client = new WebClient();
        client.DownloadDataCompleted += new
DownloadDataCompletedEventHandler(DownloadDataCallback);
        client.DownloadDataAsync(new Uri(url));
        Console.ReadLine(); // Keep the console open
    }

    private static void DownloadDataCallback(object sender,
DownloadDataCompletedEventArgs e)
    {
        if (e.Error != null)
        {
            Console.WriteLine("Error downloading data.");
        }
        else
        {
            Console.WriteLine("Download complete. Number of
bytes received: " + e.Result.Length);
        }
    }
}
```

Event-based programming pattern (EAP)

The Event-based Asynchronous Pattern (EAP) provides a way to handle asynchronous operations using events. Methods usually include the `Async` suffix, and completion is indicated by an event that typically ends with `Completed`.

- EAP uses events to handle the completion of asynchronous operations. Methods typically end with the suffix `Async`, and completion is handled by subscribing to an event that ends in `Completed`.
- Example methods include `WebClient.DownloadStringAsync` and the corresponding `DownloadStringCompleted` event.

When to Use:

- EAP is suitable when working with APIs that expose asynchronous operations through events, especially in scenarios involving GUI applications where operations need to update the UI upon completion.
- It's used in older frameworks or in applications where event-driven design is preferable for handling the results of asynchronous operations.

Example:

```
using System;
using System.Net;

public class EapExample
{
    public static void Main()
    {
        WebClient client = new WebClient();
        client.DownloadStringCompleted += (sender, e) =>
        {
            if (e.Error != null)
                Console.WriteLine("Error: " + e.Error.Message);
            else
```

```
                Console.WriteLine("Downloaded data: " +
e.Result);
        };
        client.DownloadStringAsync(new
Uri("http://example.com"));
        Console.ReadLine(); // Keep the console open
    }
}
```

Task-based asynchronous pattern (TAP)

Introduced in .NET 4.0, the Task-based Asynchronous Pattern (TAP) utilizes `Task` and `Task<T>` from the `System.Threading.Tasks` namespace. This approach is now the recommended way to handle asynchronous operations in C#.

- TAP is the recommended approach in modern .NET applications. It uses the `Task` or `Task<T>` types from the `System.Threading.Tasks` namespace. Asynchronous methods are suffixed with `Async`, and results are typically awaited using the `await` keyword.
- Example methods include `HttpClient.GetStringAsync` and any method that returns a `Task` or `Task<T>`.

When to Use:

- TAP should be used in all new development unless there is a specific reason to use an older model. It integrates seamlessly with C#'s `async` and `await` features, making it easier to write, read, and maintain asynchronous code.
- It's particularly useful in applications that perform a lot of I/O operations or require scalability and responsiveness, such as web applications and services.

Example of using TAP without async and await:

```
using System;
using System.IO;
using System.Net;
using System.Threading.Tasks;

public class TapExample
{
    public static void Main()
    {
        Task<string> task =
DownloadStringAsync("http://example.com");
        task.ContinueWith(t =>
        {
            if (t.Exception != null)
                Console.WriteLine("Error: " +
t.Exception.Message);
            else
                Console.WriteLine("Downloaded data: " +
t.Result);
        });
        Console.ReadLine(); // Keep the console open
    }

    private static Task<string> DownloadStringAsync(string url)
    {
        WebClient client = new WebClient();
        return client.DownloadStringTaskAsync(new Uri(url));
    }
}
```

Async and await (C# 5.0 and later)

With C# 5.0, the `async` and `await` keywords were introduced, greatly simplifying asynchronous programming by allowing developers to write code that looks synchronous while it performs asynchronously.

Example using `async` and `await`:

```
using System;
using System.IO;
using System.Net;
using System.Threading.Tasks;
```

```
public class AsyncAwaitExample
{
    public static async Task Main()
    {
        try
        {
            string result = await
DownloadStringAsync("http://example.com");
            Console.WriteLine("Downloaded data: " + result);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }
        Console.ReadLine(); // Keep the console open
    }

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

Making the endpoint asynchronous

When a method is converted to async, it typically involves changing its return type to `Task` or `Task<T>` and using the `await` keyword within the method to call other async methods.

This change often requires that any methods calling this newly asynchronous method must themselves handle the method asynchronously. This leads to changes in those methods as well, propagating all the way up through the layers of the application, from the lowest level (e.g., data access) to the user interface or entry point of the application.

Let's assume we have implemented the endpoint and the request is flowing through a few layers.

1. Data access layer

```
public User GetUser(int id)
{
    // Synchronous database access
    return dbContext.Users.Find(id);
}
```

2. Service layer

```
public User GetUserById(int id)
{
    return userRepository.GetUser(id);
}
```

3. Controller

```
public IActionResult GetUser(int id)
{
    var user = userService.GetUserById(id);
    return Ok(user);
}
```

Now let's change the endpoint to async version.

1. Data access layer

```
public async Task<User> GetUserAsync(int id)
{
    // Asynchronous database access
    return await dbContext.Users.FindAsync(id);
}
```

2. Service layer

```
public async Task<User> GetUserByIdAsync(int id)
{
    return await userRepository.GetUserAsync(id);
}
```

3. Controller

```
public async Task<IActionResult> GetUserAsync(int id)
{
    var user = await userService.GetUserByIdAsync(id);
    return Ok(user);
}
```

How await/async is handled under the hood?

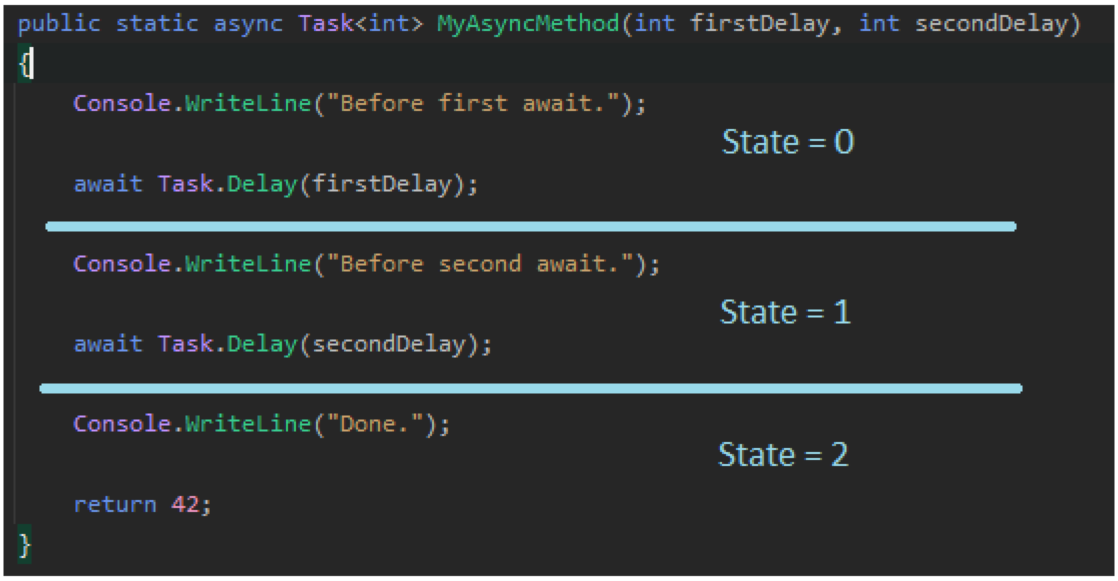
The state machine concept used in C# for handling `async` and `await` is a fundamental part of how the C# compiler manages asynchronous operations without blocking threads. When you write an `async` method that uses `await`, the C# compiler transforms your method into a state machine. This transformation enables the method to pause and resume its execution around `await` calls without blocking the thread on which it's running.

When you apply the `async` keyword to a method, the C# compiler essentially rewrites your method into a class that implements a state machine. This state machine manages the execution state of the method across multiple continuations that happen as a result of `await` operations.

More details about different ways to handle async/await:
https://devblogs.microsoft.com/dotnet/how-async-await-really-works/

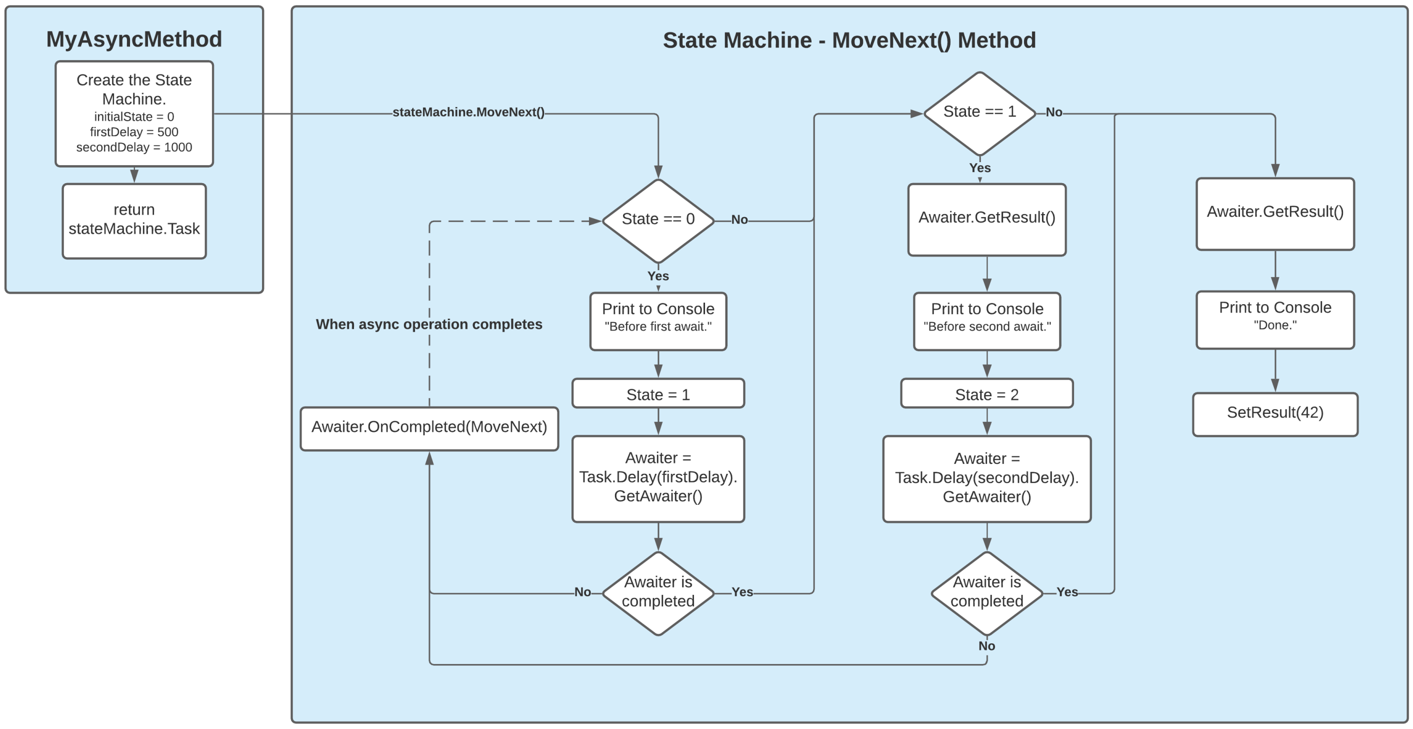
Example:

```
public static async Task<int> MyAsyncMethod(int firstDelay, int
secondDelay)
{
        Console.WriteLine("Before first await.");

        await Task.Delay(firstDelay);
        Console.WriteLine("Before second await.");

        await Task.Delay(secondDelay);

        Console.WriteLine("Done.");

        return 42;
}
```



```
public static async Task<int> MyAsyncMethod(int firstDelay, int secondDelay)
{
    Console.WriteLine("Before first await.");

    await Task.Delay(firstDelay);

    Console.WriteLine("Before second await.");

    await Task.Delay(secondDelay);

    Console.WriteLine("Done.");

    return 42;
}
```

Example of the generated state machine:



MyAsyncMethod
Create the State Machine.
initialState = 0
firstDelay = 500
secondDelay = 1000
return stateMachine.Task
stateMachine.MoveNext()
State Machine - MoveNext() Method
State == 0
No
Yes
Print to Console "Before first await."
State = 1
Awaiter = Task.Delay(firstDelay).GetAwaiter()
Awaiter is completed
No
Yes
When async operation completes
Awaiter.OnCompleted(MoveNext)
State == 1
No
Yes
Awaiter.GetResult()
Print to Console "Before second await."
State = 2
Awaiter = Task.Delay(secondDelay).GetAwaiter()
Awaiter is completed
Yes
No
Awaiter.GetResult()
Print to Console "Done."
SetResult(42)

ThreadPool

The concept of a thread pool is an essential feature in many programming environments, including .NET, which helps manage and optimize the usage of threads in a multithreaded application. A thread pool is essentially a collection of pre-instantiated, idle threads which can be used to perform multiple tasks without the overhead of creating and destroying threads each time there's a need for a new thread.

Why Use a ThreadPool?

- **Performance:** Creating and destroying threads for each task can be resource-intensive and time-consuming. By reusing existing threads from the pool, the application minimizes the overhead associated with thread management.
- **Resource Management:** Thread pools limit the number of threads that can run concurrently, thereby managing the resource usage effectively. This prevents scenarios where too many threads compete for system resources and degrade performance.

How It Works

- When a task is ready to run, it is dispatched to a thread in the thread pool. If all threads are busy, the task waits in a queue until a thread becomes available. Once a thread completes a task, it returns to the pool and becomes available for another task.
- In environments like .NET, the thread pool dynamically adjusts the number of threads it contains, based on the workload and available system resources.

In .NET, the `System.Threading.ThreadPool` class manages a pool of worker threads that execute asynchronous callbacks, typically represented as `ThreadPool.QueueUserWorkItem` calls. Here’s how it’s typically used:

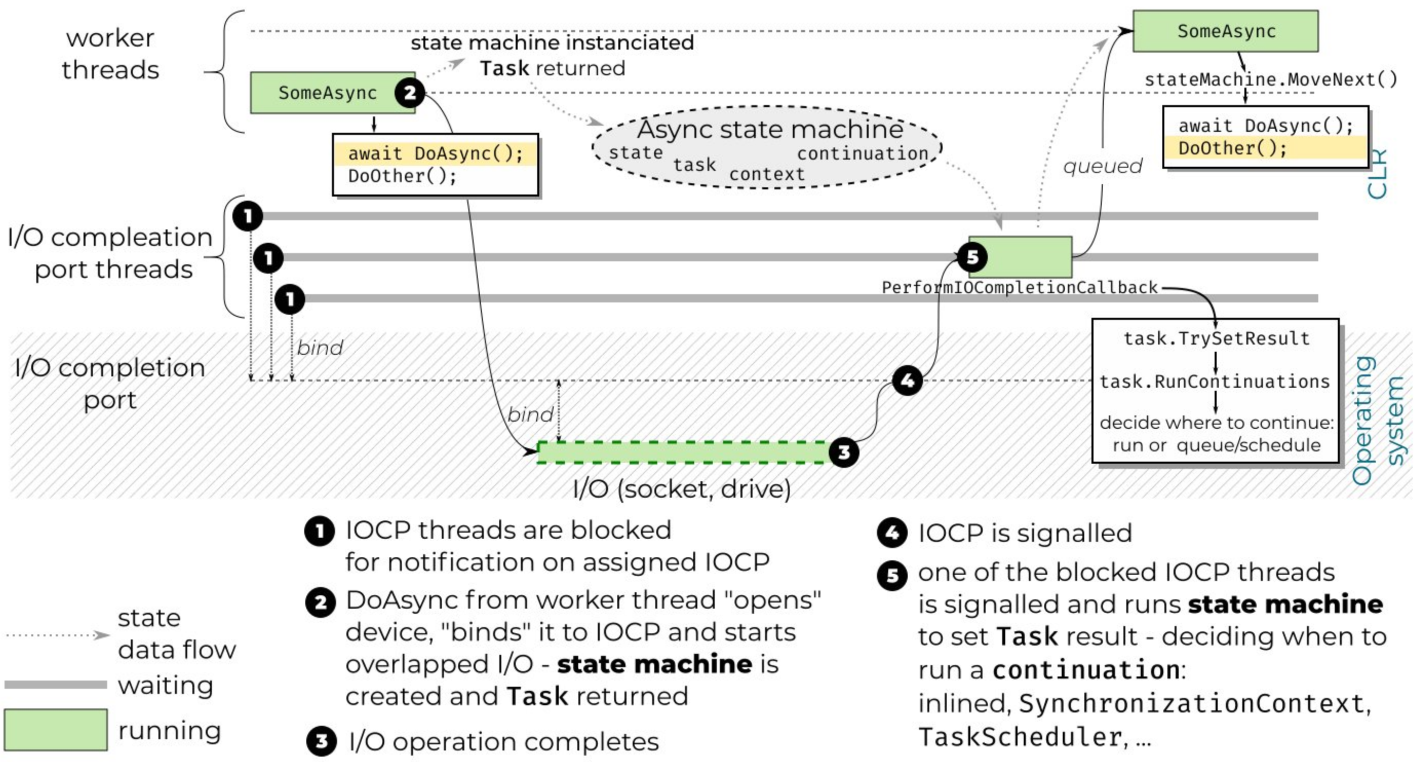
We do not have to manually create Thread:

```
Thread t=new Thread(...);
```

We can use ThreadPool:

```
using System;
using System.Threading;

public class ThreadPoolExample
{
    public static void Main()
    {
        ThreadPool.QueueUserWorkItem(ThreadProc);
        Console.WriteLine("Main thread does some work, then
sleeps.");
        Thread.Sleep(1000);

        Console.WriteLine("Main thread exits.");
    }

    static void ThreadProc(Object stateInfo)
    {
        Console.WriteLine("Task running on a thread from
threadpool.");
    }
}
```

ThreadPool and async/await in single image:



❶ IOCP threads are blocked for notification on assigned IOCP

❷ DoAsync from worker thread "opens" device, "binds" it to IOCP and starts overlapped I/O - **state machine** is created and Task returned

❸ I/O operation completes

❹ IOCP is signalled

❺ one of the blocked IOCP threads is signalled and runs **state machine** to set Task result - deciding when to run a **continuation**: inlined, SynchronizationContext, TaskScheduler, ...

https://twitter.com/konradkokosa/status/1265202184129519616

ASP.NET and the .NET ThreadPool

1. Handling Requests:

- When an HTTP request arrives at an ASP.NET application, it is assigned to a thread from the .NET thread pool. This thread is responsible for executing all the necessary operations to process the request, such as running controller actions in MVC, invoking middleware in ASP.NET Core, or executing page lifecycle events in Web Forms.
- Using a thread pool thread means that the overhead of creating and destroying a thread for each request is avoided, significantly improving performance, especially under high loads.

Complex database operations

In many business scenarios, a single high-level operation—such as completing an order, updating a user profile, or processing a financial transaction—can require multiple insert, update, or delete operations across various tables in a database.

This complexity is typically due to the relational nature of databases where data is normalized and spread across multiple tables to avoid redundancy and maintain data integrity.

Transaction

A transaction in a relational database is a sequence of operations performed as a single logical unit of work. A transaction has four primary characteristics, commonly known by the acronym ACID, which stands for Atomicity, Consistency, Isolation, and Durability.

Types of transactions:**

- Auto-commit transactions
- Implicit transactions
- Explicit transactions

Auto-commit transactions

- The auto-commit transaction mode is the default transaction mode of the SQL Server. In this mode, each SQL statement is evaluated as a transaction by the storage engine.
- In this context, if any SQL statement completes its execution successfully it is committed, and the data modification will become permanent in the database.
- On the other hand, if any statement faces any error, it will be rolled back. In this transaction mode, we don’t try to manage transactions and all operations are managed by the SQL Server.

Implicit transactions

- In the implicit transaction mode, SQL Server takes the responsibility for beginning the transactions implicitly but it waits for the commit or rollback commands from the user.
- In the implicit transaction mode, the database objects involved in the transaction will remain locked until the commit or rollback commands are executed.
- In order to use the implicit transaction mode, we need to set implicit transaction mode to ON. We can use the following syntax to enable or disable the implicit transaction mode.

Explicit transactions

- In the explicit transaction mode, we have to define the starting and ending points of the transactions.
- It means that all transactions must start with the BEGIN TRANSACTION statement and end with either COMMIT TRANSACTION or ROLLBACK TRANSACTION statements.

Using explicit transactions to keep the database in consistent state

In the example below we are explicitly using the transaction to keep the database in consistent state.

```
public async Task<int> ChangeAnimal()
{
    using var con = new SqlConnection("Data Source=db-
mssql;Initial Catalog=pegago;Integrated Security=True");
    using var com = new SqlCommand("select * from animal",
con);

    await con.OpenAsync();
    DbTransaction tran = await con.BeginTransactionAsync();
    com.Transaction = (SqlTransaction)tran;

    try
    {
        var list = new List<Animal>();
        using (var dr = await com.ExecuteReaderAsync())
        {
            while (await dr.ReadAsync())
            {
                list.Add(new Animal
                {
                    Name = dr["Name"].ToString(),
                    Description = dr["Description"].ToString()
                });
            }
        }

        com.Parameters.Clear();
        com.CommandText = "UPDATE Animal SET Name=Name+'a'
WHERE Name=@Name";
        com.Parameters.AddWithValue("@Name", list[0].Name);
        await com.ExecuteNonQueryAsync();

        //throw new System.Exception("Error");

        com.Parameters.Clear();
        com.Parameters.AddWithValue("@Name", list[1].Name);
        await com.ExecuteNonQueryAsync();

        await tran.CommitAsync();
    }
    catch (SqlException exc)
    {
        //...
        await tran.RollbackAsync();
    }
    catch (Exception exc)
    {
        //...
        await tran.RollbackAsync();
    }
}
```

Using stored procedure

Example of using stored procedure.

```
public async Task<IEnumerable<Animal>> GetAnimalsByAsync()
{
    using var con = new SqlConnection("Data Source=db-
mssql;Initial Catalog=pgago;Integrated Security=True");
    using var com = new SqlCommand("GetAnimals", con)
    {
        CommandType = CommandType.StoredProcedure
    };

    await con.OpenAsync();
    var result = new List<Animal>();
    using (var dr = await com.ExecuteReaderAsync())
    {
        while (await dr.ReadAsync())
        {
            result.Add(new Animal
            {
                Name = dr["Name"].ToString(),
                Description = dr["Description"].ToString()
            });
        }
    }
    return result;
}
```

Pros of stored procedures

- Performance benefits
- Pre-parsed SQL
- Pre-generated query execution plan
- Reduced network latency
- Potential cache benefits
- Unless using the dynamic SQL – we are protected against SQL Injection
- Allows us to control the access to db objects on very low level

Cons of stored procedure

- Our code will become highly dependent on the exact database type we are using.
- The business logic of our application may get scattered between different places – our code, and the database
- Stored procedures are harder to version control
- Our developers need to know the specific dialect of SQL
- Stored procedures may become a problem when we want to use load balancing (especially master-master replication).

Pros of raw SQL approach

- All the business logic is located within our code.
- We can more easily version control the application.
- By using specific additional libraries we can make our code more independent from the database we are using (we will see that in future classes).

Cons of raw SQL approach

- Requires a lot of boilerplate code (unless we are using something more abstract than SqlConnection/SqlCommand).

- Impedance mismatch problem
- We have to remember about SQL injection