Real-World Scenarios of Using Multithreading in ASP.NET Core

MultiThreading in C#

In simple terms

Keivan Damirchi

Include:

What is MultiThreading?

Multi-threading refers to a programming technique that enables a single process to run multiple threads or concurrent paths of execution simultaneously within a single program.

In other words, it allows a program to perform multiple tasks concurrently within a single program.

Each thread operates independently, with its own set of instructions, data stack, and other related resources. By running multiple threads, a program can take advantage of multi-core processors and achieve significant performance gains.

Remember
A process represents an application whereas a thread represents a module of the application

Example to Understand Threading

Creates two threads, each with a separate method to execute concurrently.

```
class Program {
    static void Main() {
        Thread thread1 = new Thread(new ThreadStart(CountUp));
        Thread thread2 = new Thread(new ThreadStart(CountDown));

        // Start the threads
        thread1.Start();
        thread2.Start();

        // Wait for the threads to finish
        thread1.Join();
        thread2.Join();

        Console.WriteLine("Finished.");
    }
```

```
//Result:
Counting up: 0
Counting down: 10
Counting up: 1
Counting down: 9
Counting up: 2
Counting down: 8
Counting up: 3
Counting down: 7
Counting up: 4
Counting down: 6
Counting up: 5
Counting down: 5
Counting up: 6
Counting down: 4
Counting up: 7
Counting down: 3
Counting up: 8
Counting down: 2
Counting up: 9
Counting down: 1
Finished.
```

After creating and starting both threads, the main thread waits for them to finish using the Join() method, which blocks until the threads complete.

Finally, the program prints "Finished." to the console to indicate that both threads have completed.

First thread calls the CountUp() method, which counts up from 0 to 9, printing each number to the console and waiting for half a second between each iteration.

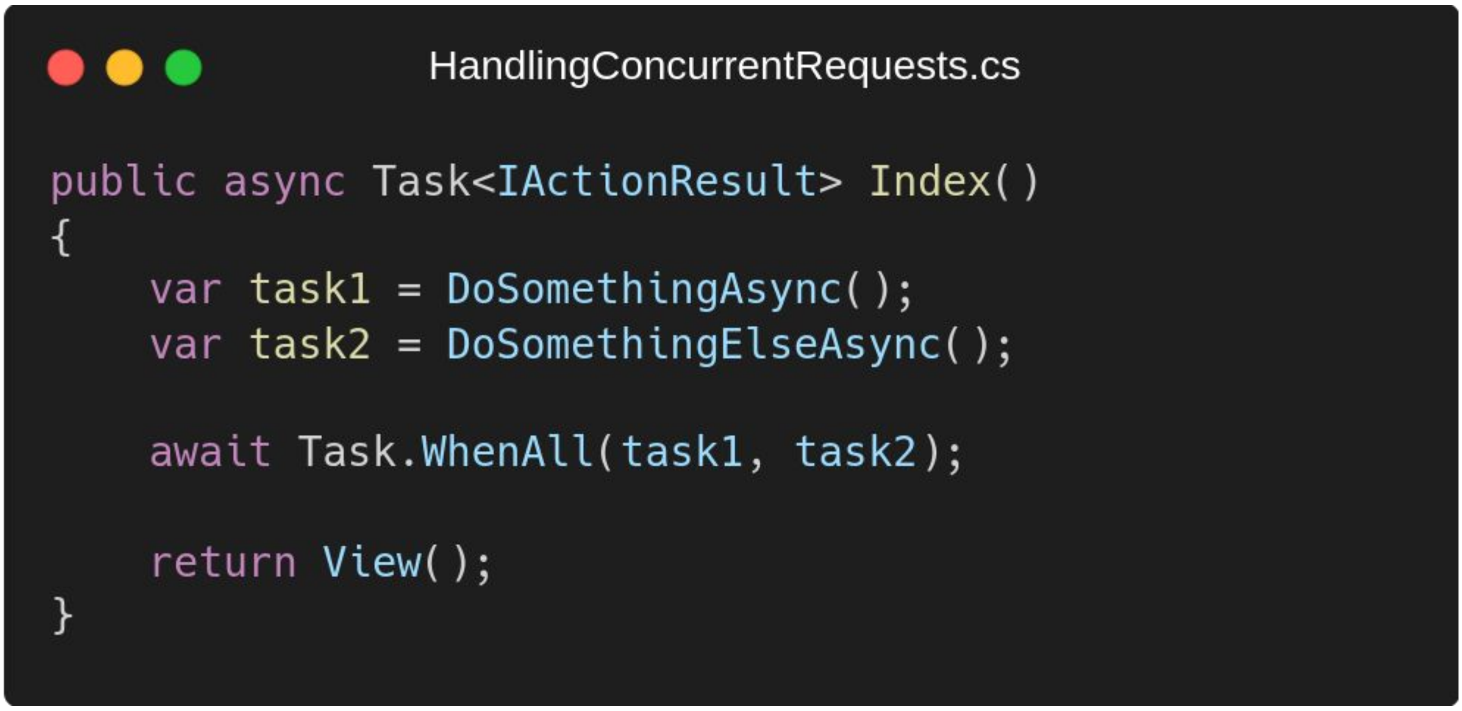
```
static void CountUp() {
    for (int i = 0; i < 10; i++) {
        Console.WriteLine("Counting up: " + i);
        Thread.Sleep(500); // Wait for half a second
    }
}
```

```
static void CountDown() {
    for (int i = 10; i > 0; i--) {
        Console.WriteLine("Counting down: " + i);
        Thread.Sleep(500); // Wait for half a second
    }
}
```

The second thread calls the CountDown() method, which counts down from 10 to 1, printing each number to the console and also waiting for half a second between each iteration.

Handling Concurrent Requests

Multithreading can be used to handle multiple requests concurrently. In ASP.NET Core, you can use the built-in Task-based Asynchronous Pattern (TAP) to create asynchronous methods that can be run in parallel.



HandlingConcurrentRequests.cs

```
public async Task<IActionResult> Index()
{
    var task1 = DoSomethingAsync();
    var task2 = DoSomethingElseAsync();

    await Task.WhenAll(task1, task2);

    return View();
}
```

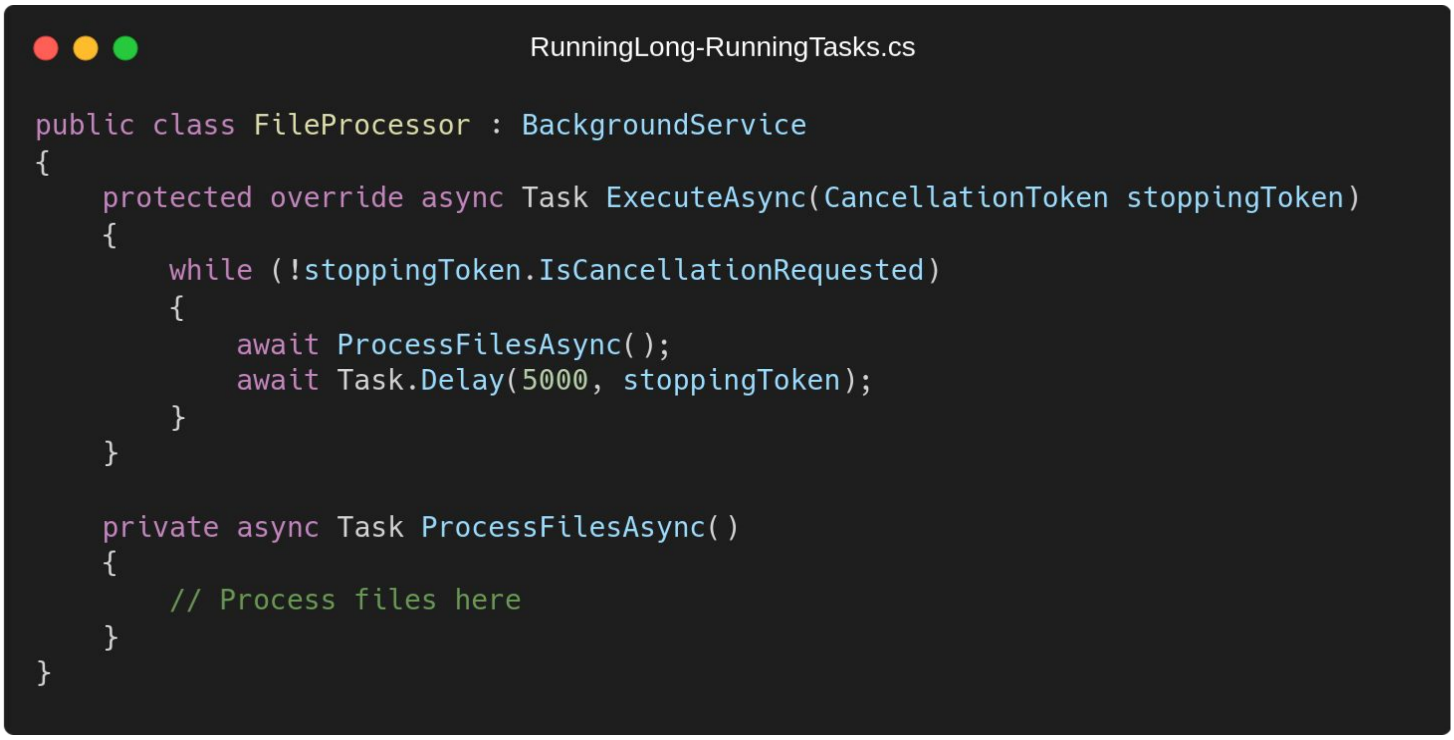
Task.WhenAll: Creates a task that will complete when all of the supplied tasks have completed.

Tips

When multiple threads are accessing shared resources, it is important to use synchronization techniques such as locks, semaphores, or atomic operations to prevent race conditions and other concurrency-related issues.

Running Long-Running Tasks

You can use multithreading to run long-running tasks in the background. For example, you can use the BackgroundService class to create a background service that runs in the background and performs a task, such as processing files.



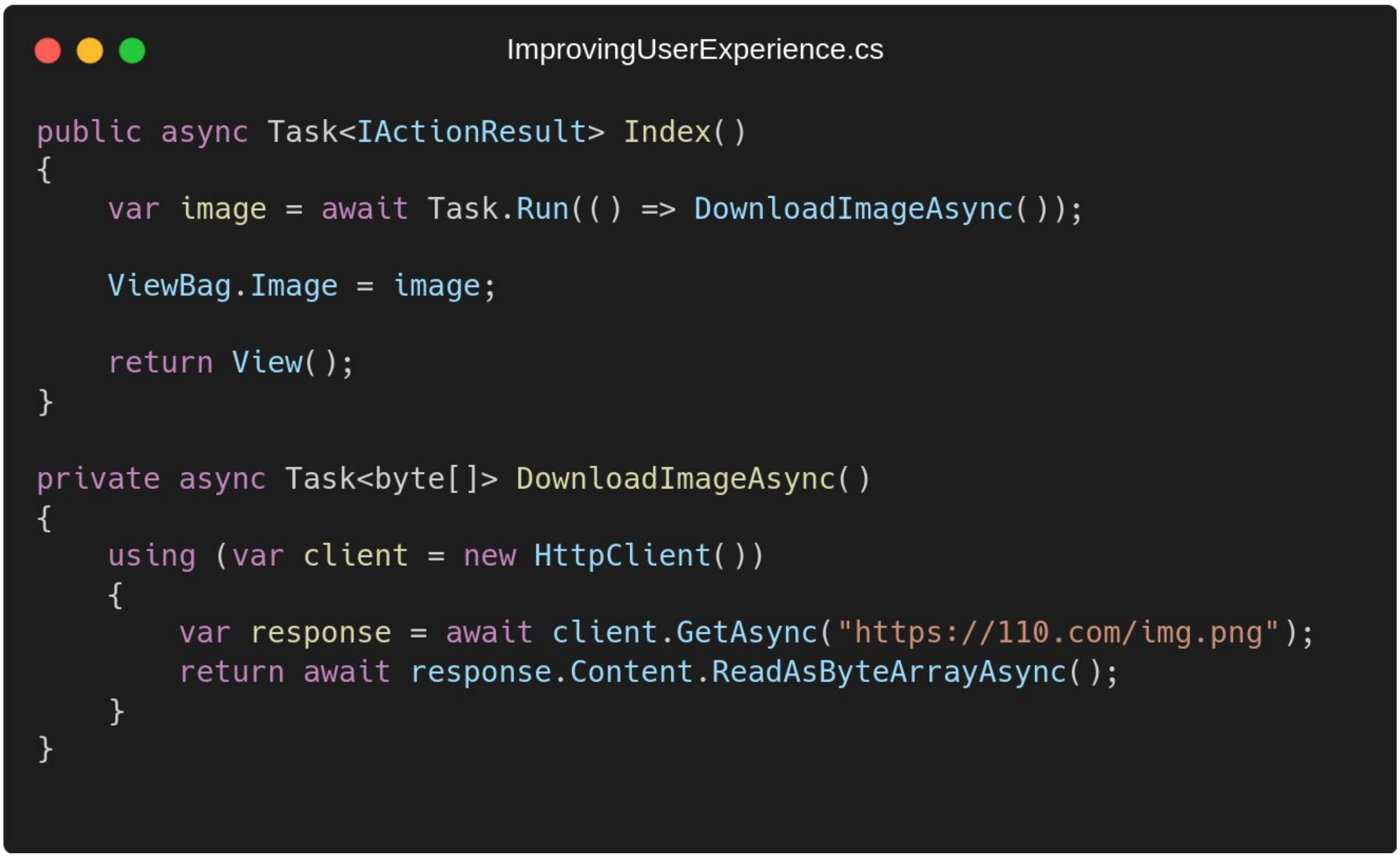
```csharp
public class FileProcessor : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            await ProcessFilesAsync();
            await Task.Delay(5000, stoppingToken);
        }
    }

    private async Task ProcessFilesAsync()
    {
        // Process files here
    }
}
```

CancellationToken: Propagates notification that operations should be canceled.

Tips
Blocking operations, such as I/O or waiting for a lock, can cause a thread to become unresponsive and degrade the performance of the program. As much as possible, use asynchronous operations or non-blocking synchronization techniques.

Improving User Experience

You can use multithreading to improve the user experience by offloading time-consuming tasks to a separate thread. For example, you can use the Task.Run method to download an image in the background while the user continues to interact with the application.



ImprovingUserExperience.cs

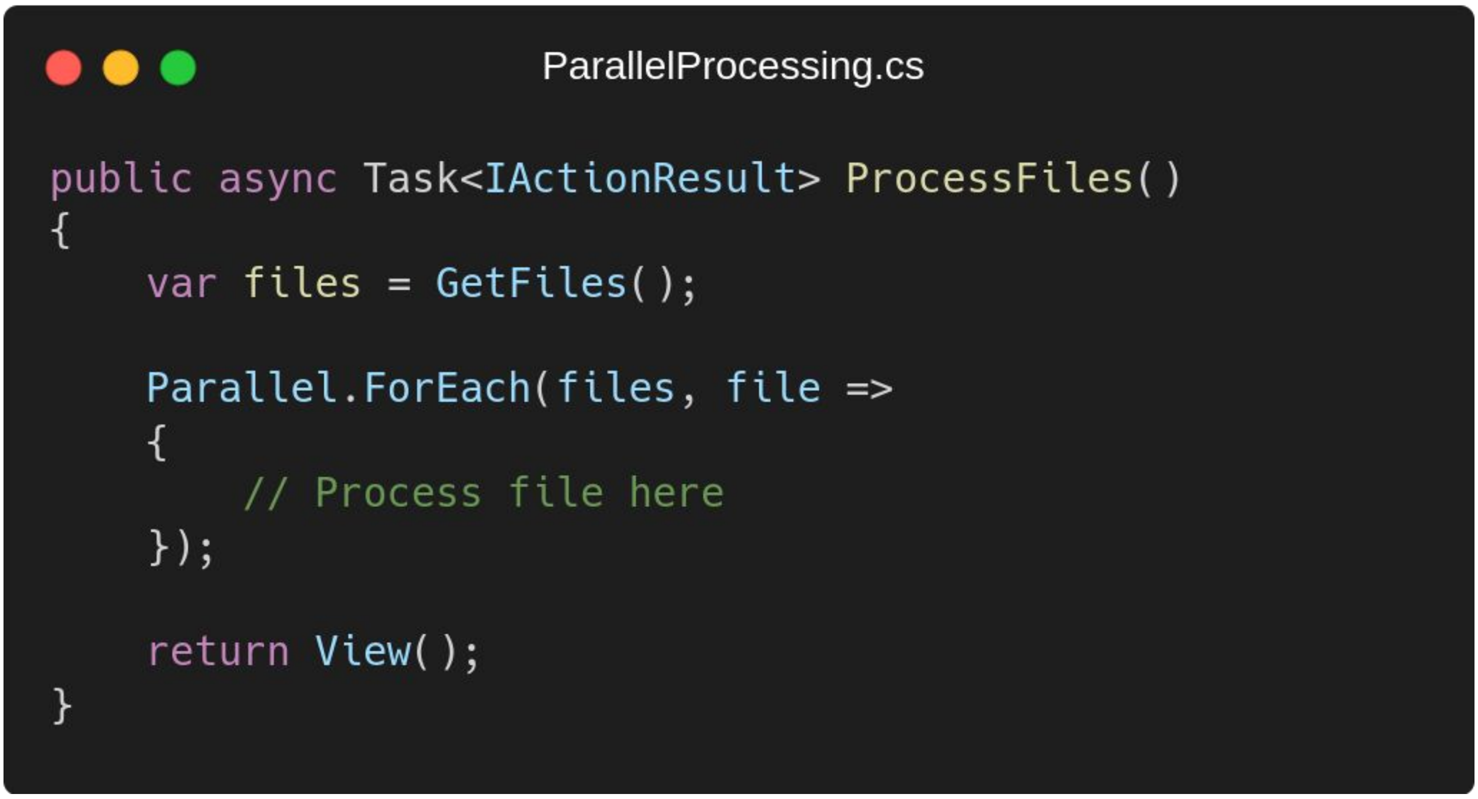
```csharp
public async Task<IActionResult> Index()
{
    var image = await Task.Run(() => DownloadImageAsync());

    ViewBag.Image = image;

    return View();
}

private async Task<byte[]> DownloadImageAsync()
{
    using (var client = new HttpClient())
    {
        var response = await client.GetAsync("https://110.com/img.png");
        return await response.Content.ReadAsByteArrayAsync();
    }
}
```

Tips

Blocking operations, such as I/O or waiting for a lock, can cause a thread to become unresponsive and degrade the performance of the program. As much as possible, use asynchronous operations or non-blocking synchronization techniques.

Parallel Processing

You can use multithreading to perform multiple tasks in parallel. For example, you can use the Parallel.ForEach method to process a list of files in parallel.



ParallelProcessing.cs

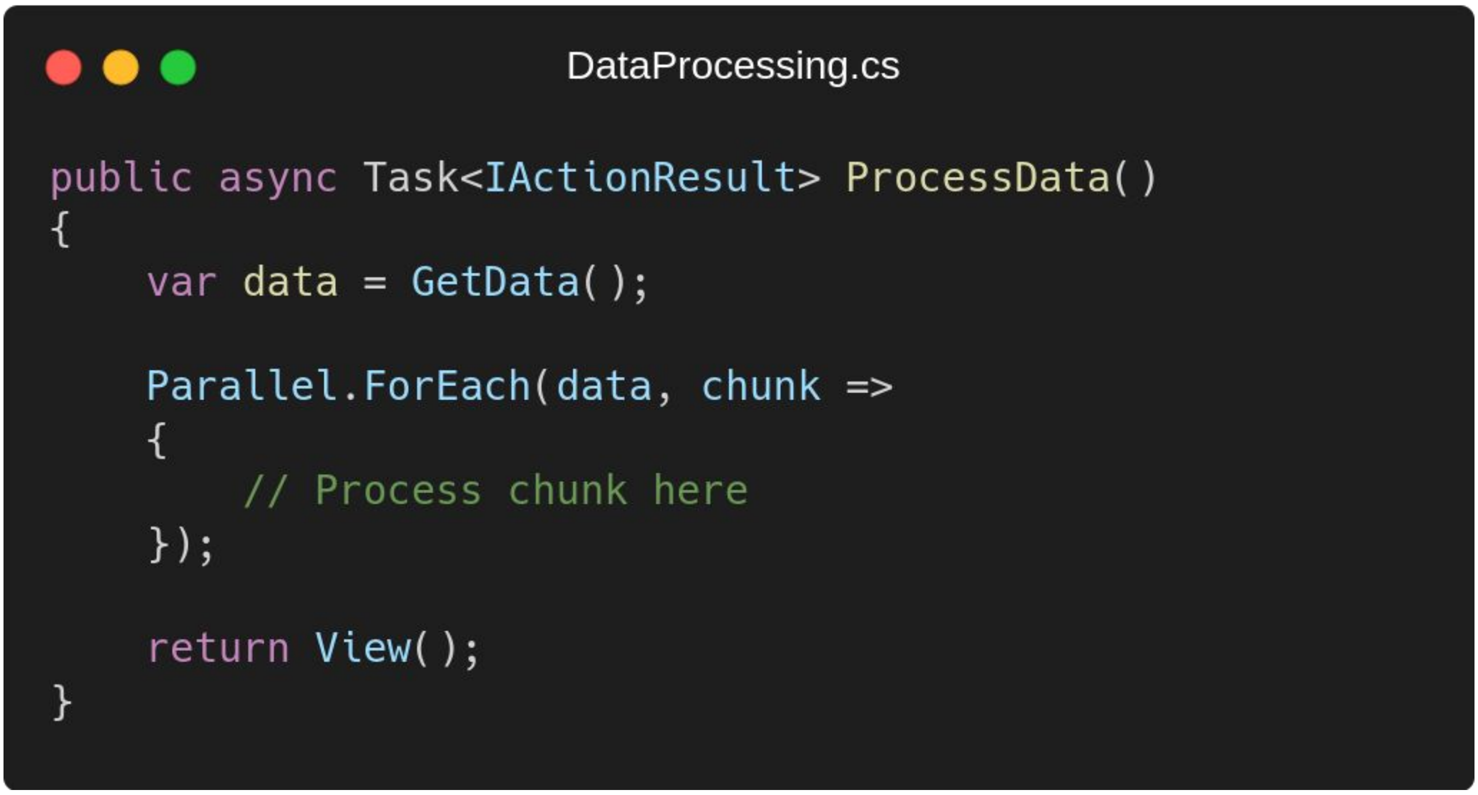
```
public async Task<IActionResult> ProcessFiles()
{
    var files = GetFiles();

    Parallel.ForEach(files, file =>
    {
        // Process file here
    });

    return View();
}
```

Parallel: Provides support for parallel loops and regions.

Tips
Thread pools are a useful way to manage the creation and reuse of threads in a program. However, creating too many threads or using long-running tasks can exhaust the thread pool and degrade the overall performance of the program.

Data Processing

You can use multithreading to divide large amounts of data into smaller chunks and process them in parallel. For example, you can use the Parallel.ForEach method to process a large dataset in parallel.


DataProcessing.cs

```
public async Task<IActionResult> ProcessData()
{
    var data = GetData();

    Parallel.ForEach(data, chunk =>
    {
        // Process chunk here
    });

    return View();
}
```

Tips
Thread starvation can occur when a high-priority thread is blocked by a low-priority thread. To avoid this, use appropriate priority levels and avoid long-running operations in high-priority threads.

Scheduling Tasks

You can use multithreading to schedule tasks to run at specific intervals or at specific times. For example, you can use the Timer class to schedule a task to run every hour.

SchedulingTasks.cs

```csharp
public class MyService
{
    private readonly Timer _timer;

    public MyService()
    {
        _timer = new Timer(14125);
        _timer.Elapsed += OnTimerElapsed;
        _timer.Start();
    }

    private async void OnTimerElapsed(object sender, ElapsedEventArgs e)
    {
        await DoSomethingAsync();
    }

    private async Task DoSomethingAsync()
    {
        // Do something here
    }
}
```

Tips

Terminating a thread abruptly can lead to data corruption or other unexpected behavior. Use safe termination techniques such as setting a flag or using a cancellation token to signal the thread to exit gracefully.

Real-Time Processing

You can use multithreading to process real-time data in real-time. For example, you can use the TPL Dataflow library to create a pipeline of processing blocks that can process real-time data.

Real-TimeProcessing.cs

```
public async Task<IActionResult> ProcessRealTimeData()
{
    var inputBlock = new BufferBlock<Data>();
    var outputBlock = new ActionBlock<Data>(data =>
    {
        // Process data here
    });

    inputBlock.LinkTo(outputBlock);

    while (true)
    {
        var data = await GetDataAsync();

        await inputBlock.SendAsync(data);
    }
}
```

BufferBlock<T>: Provides a buffer for storing data for a Dataflow.

ActionBlock<TInput>: Provides a dataflow block that invokes a provided Action<T> delegate for every data element received.

Tips
Multi-threaded programs can be difficult to test and debug. Use unit testing and profiling tools to identify and fix concurrency-related issues.

Life Cycle of a thread

In many texts, maximum 5 following situations are considered for an Thread:

Unstarted State | Runnable State | Running State | Not Runnable State | Dead State

But these situations can be expanded to more cases. Contains:

Unstarted: This is the initial state of a thread when it is created, but its Start() Method has not yet been called.

Ready: A thread is in this state when it has been created and its Start() method has been called, but it is waiting for a processor to become available to run.

Running: A thread enters this state when the operating system has assigned a processor to execute its code.

Waiting: A thread enters this state when it is blocked, waiting for a particular event to occur, such as input/output or synchronization.

Suspended: A thread enters this state when it is temporarily stopped by the operating system or another thread, but it can be resumed later.

Aborted: A thread enters this state when its Abort() method is called, indicating that it should be terminated.

Stopped: A thread enters this state when its code has finished executing or when an unhandled exception occurs.

When to Thread and When not to Thread

When to Thread

Long-running operations: Threads can be used to offload long-running operations from the main thread, allowing the user interface to remain responsive while the operation runs in the background.

Parallel processing: Threads can be used to process multiple pieces of data simultaneously, such as in image or video processing.

CPU-bound operations: Threads can be used to maximize the utilization of CPU resources for CPU-bound operations, such as scientific computations or cryptography.

Concurrency: Threads can be used to implement concurrency in systems that require multiple threads to run simultaneously, such as web servers or database systems.

When to Thread and When not to Thread

When not to Thread

Shared resources: If an operation requires access to shared resources, such as global variables or shared data structures, threading may introduce concurrency-related issues that are difficult to debug and fix.

Synchronization overhead: Threading can introduce synchronization overhead, such as locks and semaphores, that can slow down the program and make it more complex to maintain.

Resource constraints: If a system has limited resources, such as a small memory or low CPU power, creating too many threads can degrade the overall performance and stability of the system.

Common Concepts Related to Threading

Thread

A class that represents a thread of execution in a program.

Mutex

A synchronization primitive that can be used to protect shared resources by allowing only one thread to access them at a time.

Timer

A class that can be used to schedule and execute code on a periodic basis.

Monitor

A synchronization primitive that can be used to coordinate access to shared resources.

Common Concepts Related to Threading

Semaphore

A synchronization primitive that can be used to limit the number of threads that can access a shared resource.

Thread Local

A class that allows a variable to be stored and accessed on a per-thread basis.

Thread Pool

A class that manages a pool of threads that can be used to execute tasks in a multi-threaded program.

Volatile

A keyword that can be used to mark a variable as volatile, indicating that it can be accessed and modified by multiple threads without causing race conditions.

Frequently Used Thread Properties

IsAlive	Returns a Boolean value indicating whether the thread is currently executing.
Name	Gets or sets the name of the thread.
Priority	Gets or sets the priority of the thread.
ThreadState	Returns the current state of the thread, such as Running, Stopped, Suspended, or Aborted.
IsBackground	Gets or sets a value indicating whether the thread is a background thread.
ManagedThreadId	Returns a unique identifier for the thread.
CurrentCulture	Gets or sets the culture information for the thread.
IsThreadPoolThread	Gets a value indicating whether the thread is a thread from the thread pool.

Frequently Used Thread Methods

Start	Returns a Boolean value indicating whether the thread is currently executing.
Join	This method waits for a thread to complete execution before continuing with the calling thread.
Sleep	This method causes the calling thread to sleep for a specified amount of time.
Abort	This method stops the execution of a thread.
Suspend	This method suspends the execution of a thread.
Resume	This method resumes the execution of a suspended thread.
ResetAbort	This method resets the abort status of a thread.
Interrupt	This method interrupts a thread that is in a waiting or sleeping state.
Yield	This method causes the calling thread to yield execution to another thread.

Close();

Keivan Damirchi