Armen Melkumyan • 1st
Technical / Solutions Architect
6mo •

🔍 TaskScheduler in Depth: Mastering Task Execution and Concurrency Control in .NET

As a .NET developer, you probably work extensively with tasks and async/await to handle concurrency. But did you know you can gain fine-grained control over how and when tasks are executed? That’s where TaskScheduler comes into play!

The TaskScheduler is the engine behind task execution, determining whether tasks run on the ThreadPool, the UI thread, or in custom scenarios. By default, .NET uses the ThreadPoolTaskScheduler, which efficiently balances task workloads. But, for specialized needs, you can create a custom TaskScheduler to control thread allocation, execution order, and concurrency.

Key Benefits of Custom Task Schedulers:
- Optimized Concurrency: Limit the number of concurrent tasks for resource-heavy operations.
- Execution Order: Enforce strict task execution ordering where needed (FIFO, LIFO).
- Resource Management: Prevent system overload by regulating thread usage.

🔷 Use Case: Custom Scheduler for Sequential Task Execution
In scenarios like background processing or queuing systems, you might need tasks to execute sequentially. Here's a simple custom TaskScheduler that ensures only one task runs at a time.

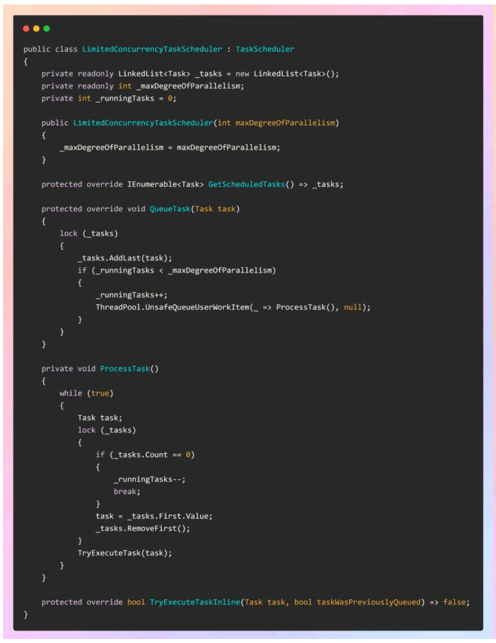
🔷 Use Case: Limiting Parallelism for CPU-bound Tasks If you want to limit concurrency to avoid CPU overuse.

Tips:
Default ThreadPool is sufficient for most use cases. Use custom schedulers only when the default behavior doesn’t meet your application’s needs.
Avoid deadlocks: Be mindful when combining custom schedulers with task continuations.

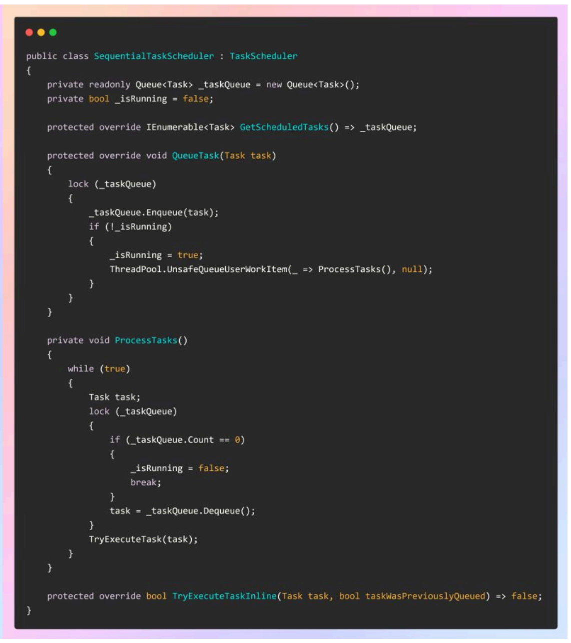
Use TaskScheduler.Current when scheduling tasks to ensure your custom scheduler is respected across the call chain.

All about TaskScheduler and more in upcoming book.

#dotnet #TaskScheduler #asyncprogramming #concurrency #dotnetcore #performanceoptimization #NewBook



```csharp
public class LimitedConcurrencyTaskScheduler : TaskScheduler
{
    private readonly LinkedList<Task> _tasks = new LinkedList<Task>();
    private readonly int _maxDegreeOfParallelism;
    private int _runningTasks = 0;

    public LimitedConcurrencyTaskScheduler(int maxDegreeOfParallelism)
    {
        _maxDegreeOfParallelism = maxDegreeOfParallelism;
    }

    protected override IEnumerable<Task> GetScheduledTasks() => _tasks;

    protected override void QueueTask(Task task)
    {
        lock (_tasks)
        {
            _tasks.AddLast(task);
            if (_runningTasks < _maxDegreeOfParallelism)
            {
                _runningTasks++;
                ThreadPool.UnsafeQueueUserWorkItem(_ => ProcessTask(), null);
            }
        }
    }

    private void ProcessTask()
    {
        while (true)
        {
            Task task;
            lock (_tasks)
            {
                if (_tasks.Count == 0)
                {
                    _runningTasks--;
                    break;
                }
                task = _tasks.First.Value;
                _tasks.RemoveFirst();
            }
            TryExecuteTask(task);
        }
    }

    protected override bool TryExecuteTaskInline(Task task, bool taskWasPreviouslyQueued) => false;
}
```



```csharp
public class SequentialTaskScheduler : TaskScheduler
{
    private readonly Queue<Task> _taskQueue = new Queue<Task>();
    private bool _isRunning = false;

    protected override IEnumerable<Task> GetScheduledTasks() => _taskQueue;

    protected override void QueueTask(Task task)
    {
        lock (_taskQueue)
        {
            _taskQueue.Enqueue(task);
            if (!_isRunning)
            {
                _isRunning = true;
                ThreadPool.UnsafeQueueUserWorkItem(_ => ProcessTasks(), null);
            }
        }
    }

    private void ProcessTasks()
    {
        while (true)
        {
            Task task;
            lock (_taskQueue)
            {
                if (_taskQueue.Count == 0)
                {
                    _isRunning = false;
                    break;
                }
                task = _taskQueue.Dequeue();
            }
            TryExecuteTask(task);
        }
    }

    protected override bool TryExecuteTaskInline(Task task, bool taskWasPreviouslyQueued) => false;
}
```

84 2 comments