Armen Melkumyan • 1st
Technical / Solutions Architect
3mo •

🚀 C# .NET Technical Interviews: Interlocked vs Lock — When to Use Which?

🔍 Interlocked: Lightweight Atomic Operations

Interlocked is ideal for atomic operations on single variables. It's fast, non-blocking, and perfect for situations where you don't need complex synchronization logic.

📌 When to Use:

Incrementing or decrementing counters.

Swapping values atomically.

Simple operations with no dependencies between steps.

Why Interlocked? It avoids the overhead of thread blocking, making it a great choice for simple atomic operations like the above.

🔍 lock: Synchronizing Complex Critical Sections

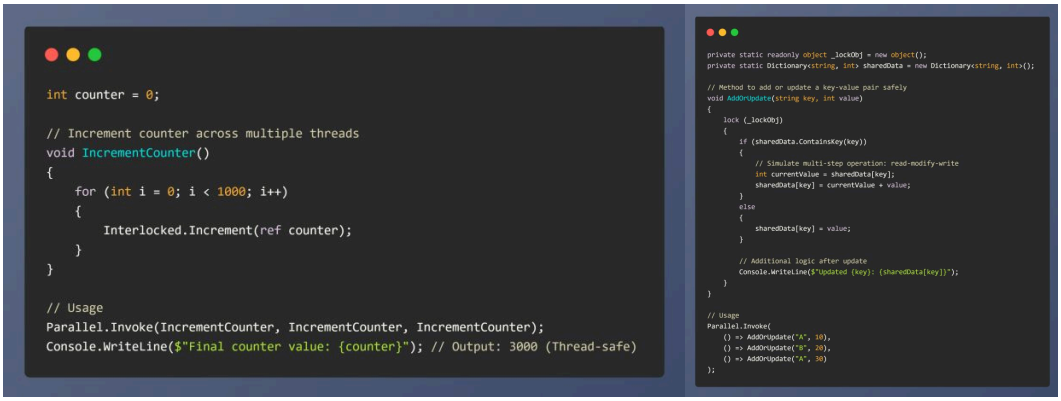
The lock keyword provides a mutual exclusion mechanism, ensuring that only one thread can execute a block of code at a time. It's best for protecting shared resources or multi-step operations where consistency is key.

📌 When to Use:

- Synchronizing access to shared resources like lists, dictionaries, or queues.
- Multi-step operations that must remain atomic.
- Avoiding race conditions in critical sections.

Why lock? It allows you to manage multi-step operations, ensuring no other thread modifies the shared resource during the operation.

#DotNet #CSharp #Concurrency #ThreadSafety #TechInterviews #PerformanceOptimization



```
int counter = 0;

// Increment counter across multiple threads
void IncrementCounter()
{
    for (int i = 0; i < 1000; i++)
    {
        Interlocked.Increment(ref counter);
    }
}

// Usage
Parallel.Invoke(IncrementCounter, IncrementCounter, IncrementCounter);
Console.WriteLine($"Final counter value: {counter}"); // Output: 3000 (Thread-safe)
```

```
private static readonly object _lockObj = new object();
private static Dictionary<string, int> sharedData = new Dictionary<string, int>();

// Method to add or update a key-value pair safely
void AddOrUpdate(string key, int value)
{
    lock (_lockObj)
    {
        if (sharedData.ContainsKey(key))
        {
            // Simulate multi-step operation: read-modify-write
            int currentValue = sharedData[key];
            sharedData[key] = currentValue + value;
        }
        else
        {
            sharedData[key] = value;
        }

        // Additional logic after update
        Console.WriteLine($"Updated {key}: {sharedData[key]}");
    }
}

// Usage
Parallel.Invoke(
    () => AddOrUpdate("A", 10),
    () => AddOrUpdate("B", 20),
    () => AddOrUpdate("A", 30)
);
```