

マルチスレッド実装を理解する

正確性とパフォーマンスのバランス

発表者：toutou (X: [@_tou_tou](#))

※ 本資料はAI (Claude) を活用して作成されています。
技術的な内容については正確性に努めていますが、誤りが含まれる可能性があります。

デモやスライドのリンク

- GitHub: [multi-thread-demo](#)
-

概要

本発表の目的

マルチスレッド環境におけるカウンター実装の課題と解決策を検討する

対象システムの要件

- 秒間数千回のメソッド呼び出し
- 複数スレッドからの同時アクセス
- システム負荷の監視指標として使用

検討する実装方式

1. 非スレッドセーフ実装
2. Lock実装
3. Lock-Free実装

1. 非スレッドセーフ実装

```
public class MethodCounter_NotThreadSafe : IMethodCounter
{
    private Dictionary<string, int> _counts = new();

    public void Record(string methodName)
    {
        if (_counts.ContainsKey(methodName))
        {
            // 非アトミックなRead-Modify-Write処理
            // この3ステップは不可分ではないため、競合状態 (Race Condition) を引き起こす。
            var count = _counts[methodName]; // 1. 読み込み (Read)
            count = count + 1;                // 2. 変更 (Modify)
            _counts[methodName] = count;      // 3. 書き込み (Write)
        }
        else
        {
            _counts[methodName] = 1;
        }
    }
}
```

問題点：カウントアップがアトミック操作でないため、競合する可能性がある

- ケース1：マルチコアでの並列実行
 - コア1のスレッドAがメモリから値 **5** を読む。ほぼ同時に、コア2のスレッドBもメモリから値 **5** を読む。
 - 両方のスレッドがそれぞれ **5 + 1** を計算し、結果の **6** をメモリに書き戻そうとする。
- ケース2：シングルコアでの並行実行（コンテキストスイッチ）
 - スレッドAがメモリから値 **5** を読んだ直後、OSがスレッドBに処理を切り替える。
 - スレッドBが値 **5** を読み、計算し、**6** を書き戻す。
 - その後、処理が戻ってきたスレッドAは、最初に読んだ **5** を元に計算してしまい、再度 **6** を上書きしてしまう。

複数スレッドでカウント

```
static async Task Step1_ShowNotThreadSafe()
{
    var counter = new MethodCounter_NotThreadSafe();
    long exceptionCount = 0;

    // 100スレッドを同時に起動
    // Task.Run() で カウントタスク を ThreadPoolのスレッドで実行する
    var tasks = Enumerable.Range(0, ThreadCount).Select(_ => Task.Run(() =>
    {
        for (int i = 0; i < CallsPerThread; i++)
        {
            try
            {
                counter.Record("TestMethod");
            }
            catch
            {
                // 例外が発生したらカウント (Dictionary破損時など)
                Interlocked.Increment(ref exceptionCount);
            }
        }
    }));

    await Task.WhenAll(tasks);
}
```

結果：データ不整合

テスト条件

- 100スレッド × 10,000回 = 合計100万回の呼び出し

```
-- 実行環境情報 --
このプロセスが利用可能な論理プロセッサ数: 24
-----

## 1. スレッドセーフでない実装の問題点
-----

100スレッドから各10,000回、合計1,000,000回の呼び出しを行います。
結果：例外が発生したり、カウントが期待値と大きくずれたりします。

-> 期待したカウント: 1,000,000
-> 実際のカウント: 117,051
-> 例外の発生回数: 1

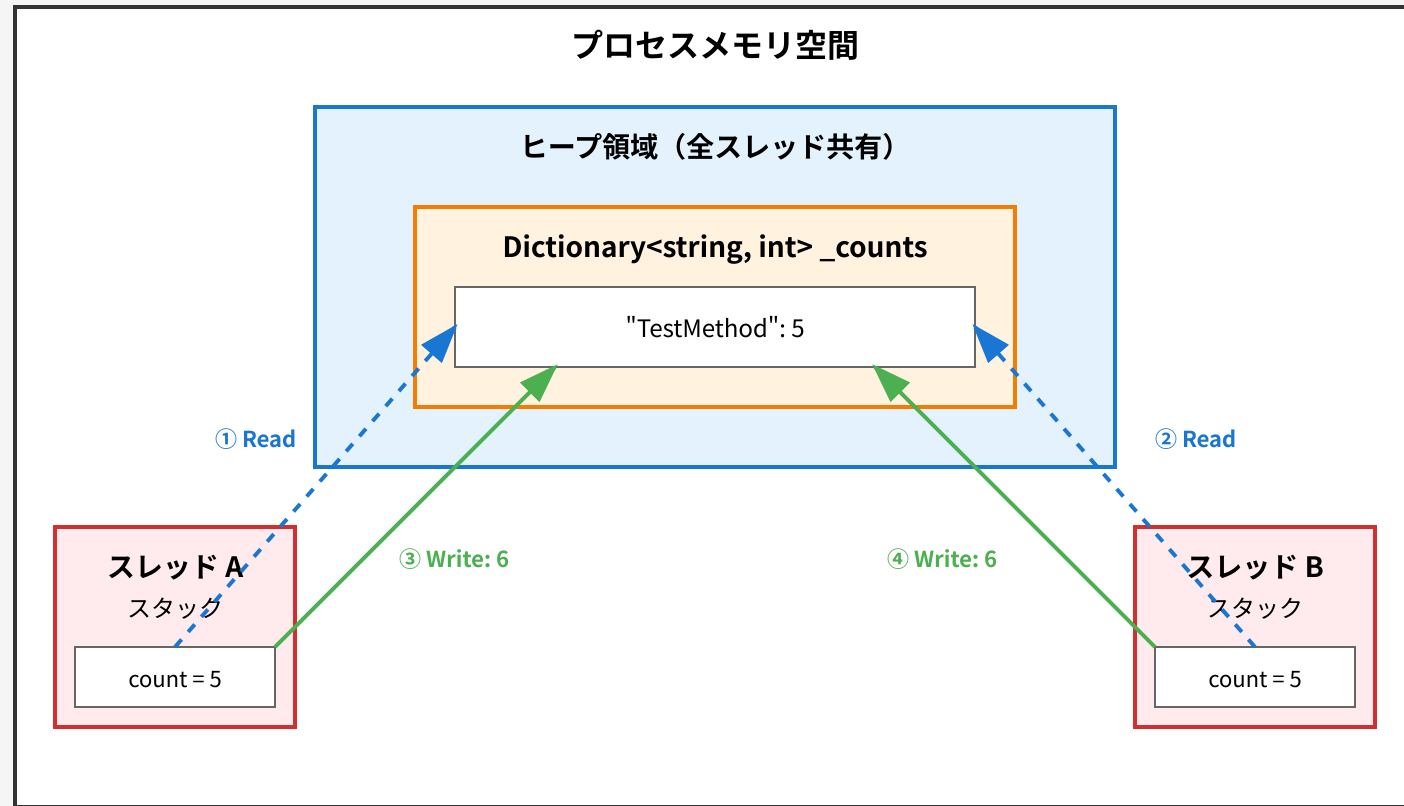
[結論] この実装はマルチスレッド環境では全く信頼できません。
```

発生した問題

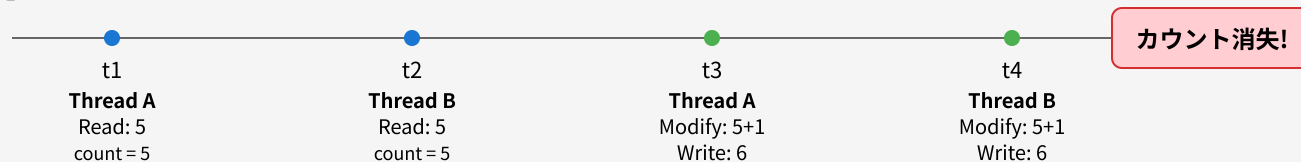
- カウント値の不整合
- Dictionaryの内部構造破損による例外

問題の原因 (1/2)

メモリ構造と競合状態の発生メカニズム



競合状態のタイムライン:



問題の原因 (2/2)

```
public class MethodCounter_NotThreadSafe : IMethodCounter
{
    private Dictionary<string, int> _counts = new();

    public void Record(string methodName)
    {
        if (_counts.ContainsKey(methodName))
        {
            // 非アトミックなRead-Modify-Write処理
            // この3ステップは不可分ではないため、競合状態 (Race Condition) を引き起こす。
            var count = _counts[methodName]; // 1. 読み込み (Read)
            count = count + 1;                // 2. 変更 (Modify)
            _counts[methodName] = count;      // 3. 書き込み (Write)
        }
        else
        {
            _counts[methodName] = 1;
        }
    }
}
```

技術的課題

1. 共有データへの非同期アクセス

- 全スレッドがヒープ上の単一Dictionaryオブジェクトを共有
- 保護機構なしで同時読み書き

2. Read-Modify-Write操作の非アトミック性

- `Record()` 内での3つの個別操作
- 操作間での割り込み発生によるカウント消失

2. Lock実装

```
public class MethodCounter_WithLock : IMethodCounter
{
    private readonly object _lock = new();
    private Dictionary<string, int> _counts = new();

    public void Record(string methodName)
    {
        lock (_lock) // 一度に1スレッドのみ実行
        {
            if (_counts.ContainsKey(methodName))
            {
                _counts[methodName]++;
                // 以下と同等
                // var count = _counts[methodName]; // 1. 読み込み (Read)
                // count = count + 1;                // 2. 変更 (Modify)
                // _counts[methodName] = count;      // 3. 書き込み (Write)
            }
            else
            {
                _counts[methodName] = 1;
            }
        }
    }
}
```

ポイント

- **カウントが正確:** lockステートメント内の相互排他により、すべての操作が順序通りに実行される

マルチスレッドでLock実装の検証

```
static async Task Step2_ShowWithLock()
{
    var counter = new MethodCounter_WithLock();

    // 100スレッドを同時に起動 (例外処理は不要)
    var tasks = Enumerable.Range(0, ThreadCount).Select(_ => Task.Run(() =>
    {
        for (int i = 0; i < CallsPerThread; i++)
        {
            counter.Record("TestMethod");
        }
    }));

    await Task.WhenAll(tasks);

    // 結果を表示 (カウントは必ず正確)
    Console.WriteLine($" -> 期待したカウント: {TotalCalls:N0}");
    Console.WriteLine($" -> 実際のカウント: {counter.GetCountsAndReset()["TestMethod"]:N0}");

    // 結論: 'lock'を使えば、安全で正確なカウンターを簡単に実装できます
}
```

- 複数スレッドで書き込み、正しくカウントされるかを確認

Lock実装の結果

```
---  
## 2. 'lock'によるスレッドセーフな実装  
---
```

同じテストを、今度は'lock'で保護した実装に対して行います。
結果：例外は発生せず、カウントは完全に正確になります。

-> 期待したカウント: 1,000,000
-> 実際のカウント: 1,000,000

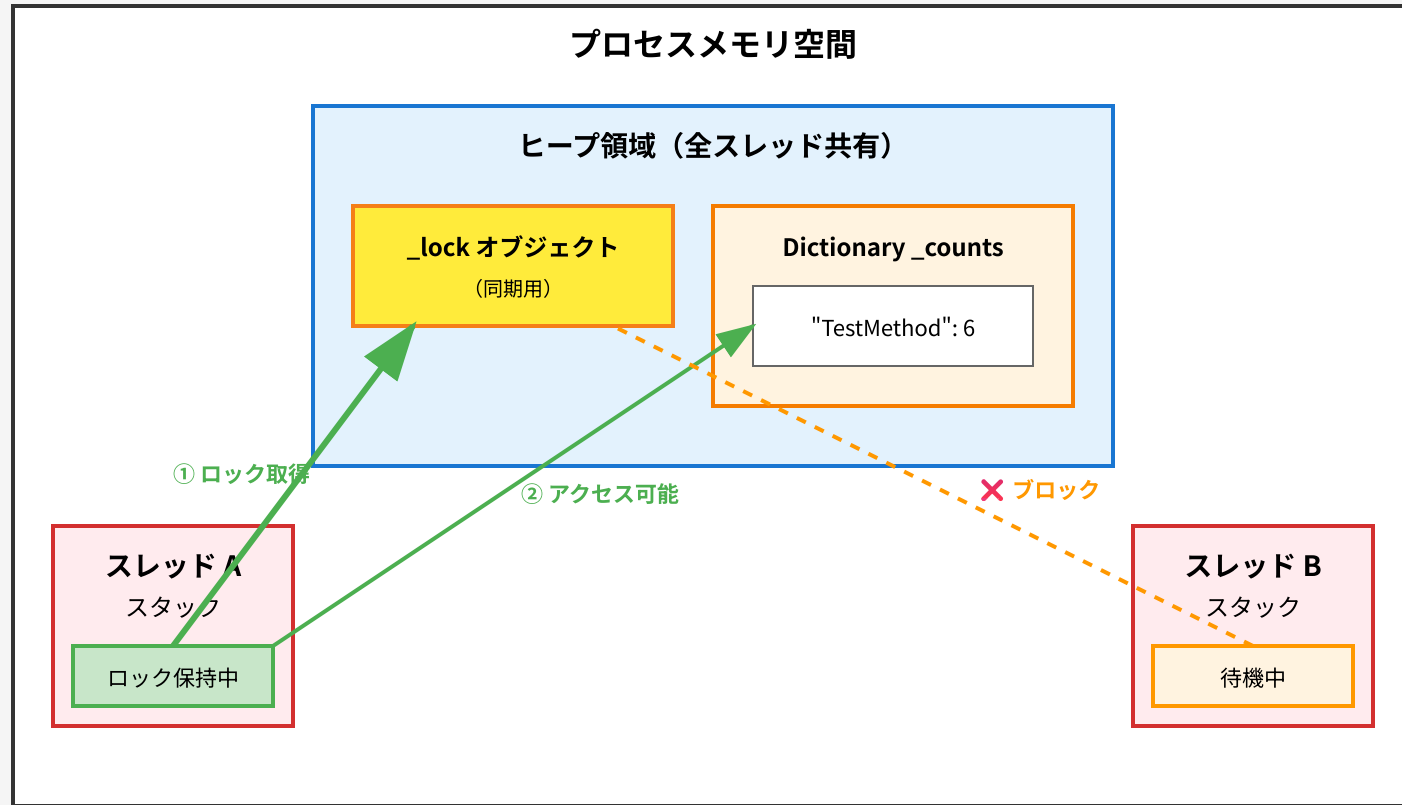
[結論] 'lock'を使えば、安全で正確なカウンターを簡単に実装できます。

結果

- カウント値の完全な一致
- 例外の発生なし

lock の仕組み (1/2)

lock使用時のメモリアクセス保証



lock の仕組み (2/2)

- lockオブジェクトの取得は CPU レベルで「アトミック操作であること」が保証されている
- 単一の割り込み不可能なCPU命令「Compare-And-Swap (CAS)」で実現される

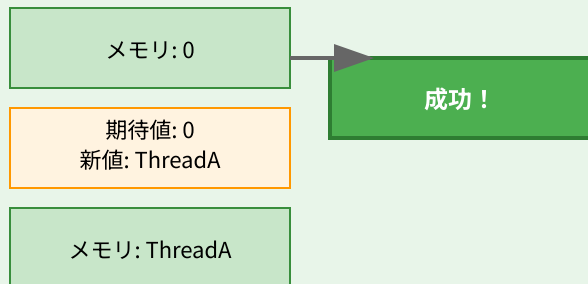
CAS (Compare-And-Swap) のアトミック動作

LOCK CMPXCHG 命令

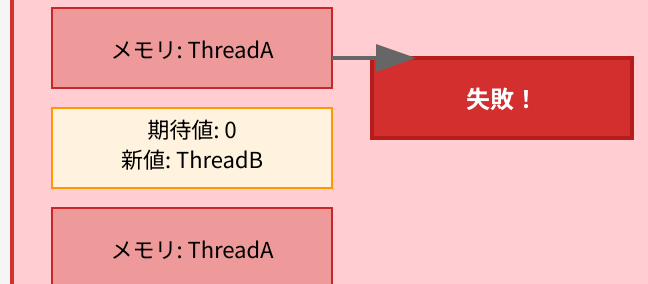
LOCK CMPXCHG [memory_address], new_value

1. 現在のメモリ値を読み込み
 2. 期待値 (0) と比較
 3. 一致すれば新しい値 (ThreadID) を書き込み
- ※ これらすべてが1つの割り込み不可能な命令として実行

成功ケース (Thread A)

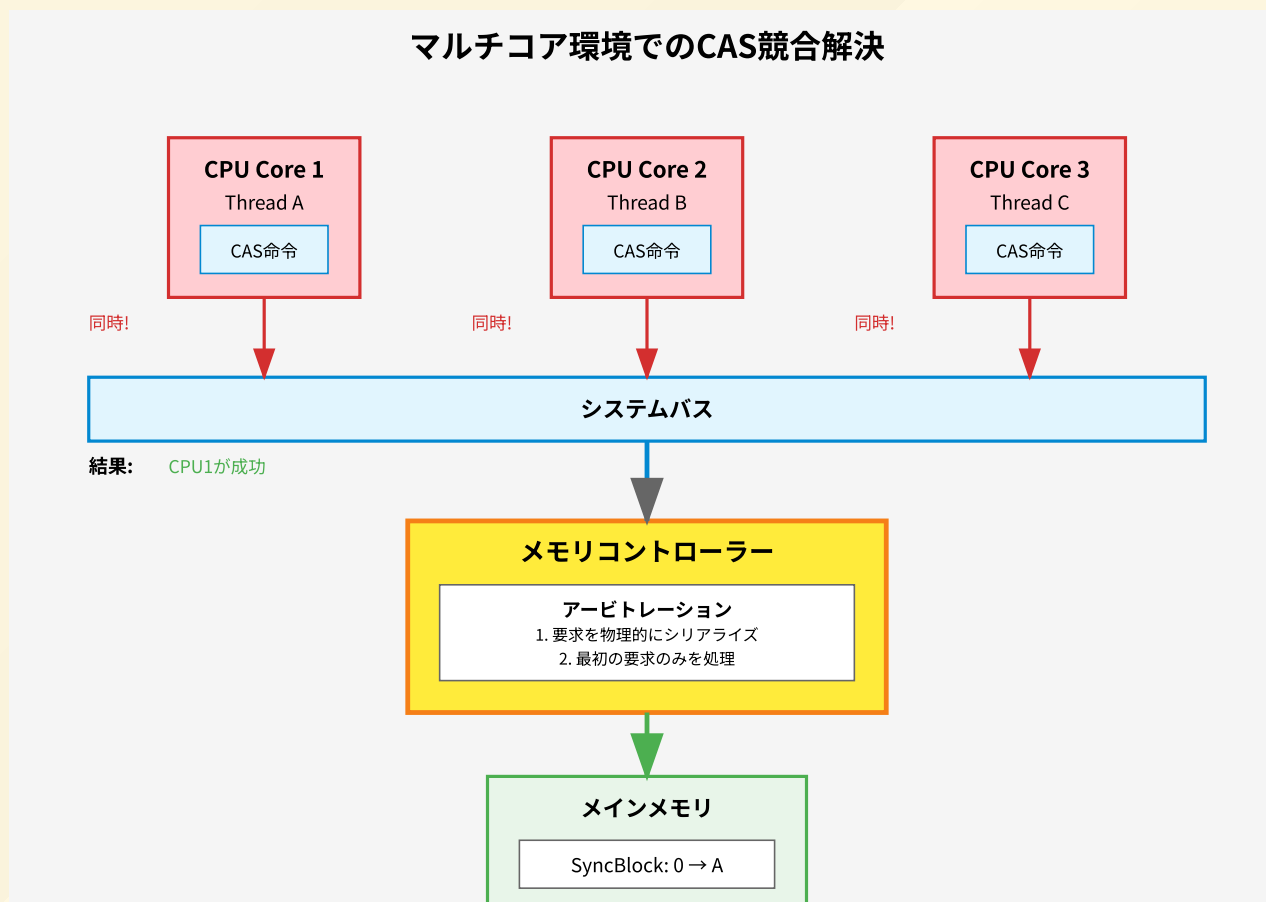


失敗ケース (Thread B)



lock取得の競合解決

- マルチコア（CPU）環境では、同時に CAS 命令を発行する可能性がある
- メモリコントローラーでメインメモリ上のlockオブジェクトを取得順がシリアルライズされる



Lock実装の懸念事項

スレッドの「lock取得の順番待ち」が
パフォーマンス低下につながる可能性

[参考] Lock実装

```
public class MethodCounter_WithLock : IMethodCounter
{
    private readonly object _lock = new();
    private Dictionary<string, int> _counts = new();

    public void Record(string methodName)
    {
        lock (_lock) // ← スレッドが順番待ちでボトルネック
        {
            if (_counts.ContainsKey(methodName))
                _counts[methodName]++;
            else
                _counts[methodName] = 1;
        }
    }

    public Dictionary<string, int> GetCountsAndReset()
    {
        // 値を読み取るときに、保持しているカウントをリセットする
        // このため lock が必要
        lock (_lock)
        {
            var result = _counts;
            _counts = new Dictionary<string, int>();
            return result;
        }
    }
}
```


3. Lock-Free実装

ConcurrentQueue<T>を使った高速化

```
public class MethodCounter_LockFree : IMethodCounter
{
    private ConcurrentQueue<string> _events = new();

    public void Record(string methodName)
    {
        _events.Enqueue(methodName); // lockなしで超高速!
        // 以下と同等
        //var currentQueue = _events; // ステップ1: 参照をコピー
        //_events = new ConcurrentQueue<string>(); // ステップ2: 新しいキューを代入
    }

    public Dictionary<string, int> GetCountsAndReset()
    {
        // Interlocked.Exchange は全CPUコア間で同期してアトミックなキュー交換を実行する
        // GetCountsAndReset での一貫性は保証するが、Recordとの競合は防げない
        var currentQueue = Interlocked.Exchange(
            ref _events, new ConcurrentQueue<string>()
        );

        // Queueにたまった要素をメソッド毎に集計
        var counts = new Dictionary<string, int>();
        while (currentQueue.TryDequeue(out var methodName))
        {
            if (counts.ContainsKey(methodName))
                counts[methodName]++;
            else
                counts[methodName] = 1;
        }
        return counts;
    }
}
```

Lock実装とLock-Free実装の比較

```
static async Task Step3_ComparePerformance()
{
    const int durationSeconds = 3;

    // --- Lock版のテスト ---
    // テスト前にGCを実行し、メモリ状態をクリーンにする
    GC.Collect();
    GC.WaitForPendingFinalizers();
    await Task.Delay(200); // 安定待ち

    var lockCounter = new MethodCounter_WithLock();
    long lockWrites = await RunWriteOnlyTest(lockCounter, durationSeconds);
    Console.WriteLine($" -> 🟢 Lock版: {lockWrites / durationSeconds,15:N0} 件/秒");

    // --- Lock-free版のテスト ---
    GC.Collect();
    GC.WaitForPendingFinalizers();
    await Task.Delay(200);

    var lockFreeCounter = new MethodCounter_LockFree();
    long lockFreeWrites = await RunWriteOnlyTest(lockFreeCounter, durationSeconds);
    Console.WriteLine($" -> ⚡ Lock-free版: {lockFreeWrites / durationSeconds,15:N0} 件/秒");

    double speedup = (double)lockFreeWrites / lockWrites;
    Console.WriteLine($"📄\n   [結論] Lock-Free版はLock版の約{speedup:F2}倍高速!");
}

// RunWriteOnlyTestヘルパーメソッド
static async Task<long> RunWriteOnlyTest(IMethodCounter counter, int durationSeconds)
{
    long totalWritten = 0;
    var cts = new CancellationTokenSource();
    var keys = Enumerable.Range(0, 100).Select(i => $"Method_{i}").ToArray();



    var writerTasks = Enumerable.Range(0, 100)
        .Select(threadIndex => Task.Run(() =>
        {
            var random = new Random(threadIndex);
            while (!cts.IsCancellationRequested)
            {
                counter.Record(keys[random.Next(keys.Length)]);
                Interlocked.Increment(ref totalWritten);
            }
        }
    )).ToList();

    await Task.Delay(TimeSpan.FromSeconds(durationSeconds));
    cts.Cancel();
    await Task.WhenAll(writerTasks);
    return totalWritten;
}
```

Lock実装とLock-Free実装の性能比較 [Windows環境]

```
-----  
## 3. パフォーマンスと Lock-Free版の 登場  
-----
```

'lock'は确实ですが、スレッドが互いを待ち合うため性能が低下する可能性があります。
そこで登場するのが Lock-Free実装です。純粋な書き込み性能を比較してみましょう。

```
->  Lock版 :           7,673,841 件/秒  
->  Lock-free版 :       46,486,868 件/秒
```



[結論] Lock-Free版は Lock版の約6.06倍、高速に書き込みが可能です！

- Thread待機時間が減ってパフォーマンスが6倍向上

Lock実装とLock-Free実装の性能比較 [WSL環境]

3. パフォーマンスとLock-Free版の登場

'lock'は确实ですが、スレッドが互いを待ち合うため性能が低下する可能性があります。そこで登場するのがLock-Free実装です。純粋な書き込み性能を比較してみましょう。

->  Lock版: 9,204,798 件/秒
->  Lock-free版: 14,737,224 件/秒

[結論] Lock-Free版はLock版の約1.60倍、高速に書き込みが可能です！

- Thread待機時間が減ってパフォーマンスが1.6倍向上

Lock-Free実装の課題: 大量の同時読み書きシナリオ

```
static async Task Step4_ShowLockFreeDataLoss()
{
    var counter = new MethodCounter_LockFree();
    //var counter = new MethodCounter_WithLock(); // Lock版
    long totalWritten = 0;
    long totalRead = 0;
    var cts = new CancellationTokenSource();

    // 大量書き込み
    var writerTasks = Enumerable.Range(0, ThreadCount).Select(_ => Task.Run(() =>
    {
        while (!cts.IsCancellationRequested)
        {
            counter.Record("Event");
            Interlocked.Increment(ref totalWritten);
        }
    })).ToList(); // タスクを開始させる (Enumerableのままだと遅延実行になってしまう)

    // 大量読み出し (リセットによる書き込みも含む)
    var readerTask = Task.Run(async () =>
    {
        while (!cts.IsCancellationRequested)
        {
            var c = counter.GetCountsAndReset();
            Interlocked.Add(ref totalRead, c.Values.Sum());
            await Task.Delay(1);
        }
    });

    await Task.Delay(10000); // 10秒間テスト
    cts.Cancel();
    await Task.WhenAll(writerTasks.Append(readerTask));
    totalRead += counter.GetCountsAndReset().Values.Sum(); // 残りを回収

    Console.WriteLine($" -> 書き込み総数: {totalWritten:N0}");
    Console.WriteLine($" -> 読み取り総数: {totalRead:N0}");
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($" -> ロストした数: {totalWritten - totalRead:N0}");
    Console.WriteLine("\n 【結論】 わずかですが、データがロストしてしまいました!\n");
    Console.ResetColor();
}
```

Lock-Free実装の大量同時書き込み結果

4. Lock-Free版の落とし穴：データロスト

Lock-Free版は高速ですが、書き込みと読み出しが特定のタイミングで重なると...
高負荷をかけながら、書き込みと読み出しを同時に実行してみます。

-> 書き込み総数: 151,192,646
-> 読み取り総数: 151,192,644
-> ロストした数: 2

[結論] わずかですが、データがロストしてしまいました！

結果

- わずかにデータロスト発生

データロストの理由

Record() と GetCountsAndReset() の競合

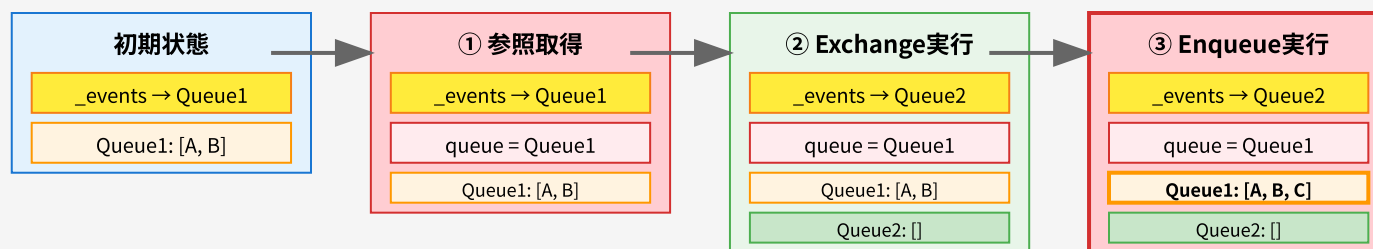
Record("MethodC") - Thread A

```
public void Record(string methodName)
{
    // ① 参照を取得
    var queue = this._events;
    // ② ここで割り込み！
    // ③ 古いキューに追加
    queue.Enqueue(methodName);
}
```

GetCountsAndReset() - Thread B

```
public Dictionary GetCountsAndReset()
{
    // ② アトミックに交換
    var oldQueue = Interlocked.Exchange(
        ref _events, new ConcurrentQueue());
    // 集計処理...
    return counts;
}
```

メモリ状態の変化



結果

GetCountsAndReset() は Queue1 を集計後に破棄

"MethodC" は集計に含まれずデータロスト！

Interlocked.ExchangeはGetCountsAndResetの内部的な一貫性は保証するが、Recordメソッドとの競合は防げない

まとめ：3つの実装の比較

| 実装方式 | 正確性 | 性能 | 適用場面 |
|---------------|-----|----|-------------------------|
| NotThreadSafe | ✗ | — | なし |
| WithLock | ✓ | ⚠ | 正確性が最優先 (課金、在庫管理とか) |
| LockFree | ⚠ | ✓ | 高速性が重要 (アクセス解析、ログとか) |

- 要件に応じて実装を使い分けよう

その他・検証したかったこと

- メモリバリアとvolatile
 - **Read Barrier**: 読み込み順序保証
 - **Write Barrier**: 書き込み順序保証
 - **Full Barrier**: 両方保証 (lock, Interlocked)
 - volatile は順序保証のみ、アトミック性は保証しない
- 同期プリミティブの使い分け
 - **lock**: 一般的な相互排他
 - **ReaderWriterLockSlim**: 読み取り頻度が >> 書き込み頻度 の場合
 - **SemaphoreSlim**: 同時アクセス数制限
- パフォーマンス考慮したlockの使い分け
 - **読み書き比率**: 7:1以上なら ReaderWriterLockSlim
 - **待機時間**: 短時間=SpinLock、長時間=通常lock
 - **スケーラビリティ**: 高負荷時はlock-freeデータ構造
- ベンチマーク実装に BenchmarkDotNet を使う

リンク

GitHub: [multi-thread-demo](#)

X: [@__tou__tou](#)