Синхронизация данных для многопоточности

Категории

Synchronizing data for multithreading
https://docs.microsoft.com/ru-ru/dotnet/standard/threading/synchronizing-data-for-multithreading

Когда вызов метода или обращение к свойству объекта может осуществляться из нескольких потоков возникает необходимость синхронизации. Поскольку один из обращающихся к объекту потоков может быть вытеснен при продолжении выполнения остальных, объект может находится в некорректном состоянии. Классы, чьи методы защищены от прерывания являются потоко-безопасными.

.NET предоставляет ряд возможностей для синхронизации доступа к членам экземпляров и статическим:

- Синхронизация участков кода. Для этого можно использовать класс Monitor или поддержку компилятором (lock) этого класса. Причем, для повышения быстродействия, рекомендуется блокировать только фрагменты кода, где это действительно нужно.
- Ручная синхронизация с помощью одного из примитивов синхронизации.
- Контексты синхронизации. Только для .NET и Xamarin приложений можно использовать SynchronizationAttribute.
- Классы коллекций в пространстве имен System.Collections.Concurrent. Эти классы предоставляют операции добавления и удаления со встроенной синхронизацией.

Среда CLR предоставляет многопоточную модель, в которой классы делятся на несколько категорий, которые можно синхронизировать различными способами в зависимости от требований. В следующей таблице показано, какая поддержка синхронизации предоставляется для полей и методов в зависимости от категории синхронизации.

Категория	Глобальные переменные	Статические поля	Статические методы	Поля экзем-пляров	Методы экзем-пляров	Блок кода
Нет синхрони-зации	–	–	–	–	–	–
Контекст син-хронизации	–	–	–	+	+	–
Синхронизация блока кода	–	–	Если указано	–	Если указано	Если указано
Ручная синхронизация	Вручную	Вручную	Вручную	Вручную	Вручную	Вручную

Нет синхронизации

Это стандартное поведение для объектов. Любой поток может получить доступ к любому методу или полю в любой момент времени.

Синхронизация блока кода

Можно использовать класс Monitor или ключевое слово lock для синхронизации блока кода, методов экземпляра и статических методов. Синхронизация для статических полей не поддерживается.

Когда код выполняется потоком, выполняется попытка получить блокировку. Если блокировка получена другим потоком, поток ожидает, пока блокировка не освободится. Когда поток выходит из синхронизируемого блока кода, блокировка освобождается в независимости от способа выхода из блока кода.

Оператор lock реализуется с использованием вызова методов Monitor.Enter и Monitor.Exit. Соответственно, другие методы класса Monitor можно использовать внутри синхронизируемого блока.

Можно также описать метод с атрибутом MethodImplAttribute со значением MethodImplOptions.Synchronized, что эквивалентно заключению всего кода метода в оператор lock.

Thread.Interrupt может использоваться для вывода потока из заблокированного состояния, например, ожидание доступа к синхронизированной области кода. Thread.Interrupt также используется для вывода потоков из таких операций, как Thread.Sleep.

Не стоит использовать в качестве объекта блокировки тип typeof(MyType) для синхронизации статических методов. Для этого следует использовать приватное статическое поле. Точно так же не стоит использовать this для синхронизации методов экземпляра. Для этого следует использовать приватное поле. Дело в том, что класс или экземпляр могут быть заблокированы не нашим кодом, что может привести к взаимоблокировкам и проблемам с производительностью. Также не следует использовать в качестве объекта блокировки строки, поскольку они могут быть интернированны.

Если в синхронизируемом блоке кода выбрасывается исключение, полученная блокировка снимается автоматически.

Ручная синхронизация и обзор синхронизирующих примитивов

Overview of synchronization primitives
https://docs.microsoft.com/ru-ru/dotnet/standard/threading/overview-of-synchronization-primitives

Платформа .NET предоставляет ряд типов для синхронизации доступа к общим ресурсам и для организации взаимодействия потоков.

Использование нескольких примитивов для защиты доступа к общему ресурсу недопустимо из-за возможного обхода блокировки.

Класс WaitHandle и облегченные типы для синхронизации

Часть примитивов синхронизации .NET наследуются от класса System.Threading.WaitHandle, который инкапсулирует нативный объект ядра операционной системы, и использует механизм сигнализирования для взаимодействия потоков. Классы:

- System.Threading.Mutex, который предоставляет эксклюзивный доступ к общему ресурсу. Он может находиться в двух состояниях: свободен (сигнальное состояние) и занят (несигнальное состояние). Свободным является только когда ни один из потоков не имеет к нему доступа.
- System.Threading.Semaphore, который ограничивает количество потоков, получающих доступ к общему ресурсу или пулу ресурсов. Семафор свободен (сигнальное состояние), когда счетчик больше нуля, и занят, когда значение равно нулю.

- System.Threading.EventWaitHandle, который представляет событие синхронизации потоков. Может находится в двух состояниях, как и мьютекс.
- System.Threading.AutoResetEvent, наследуемый от EventWaitHandle, автоматически сбрасывается в несигнальное состояние (занят) после освобождения одного ожидающего потока при получении сигнала.
- System.Threading.ManualResetEvent, наследуемый от EventWaitHandle, при получении сигнала остается в сигнальном состоянии, пока не будет вызван метод Reset.

В .NET Framework, .NET Core и .NET 5+ некоторые из этих типов могут представлять именованные дескрипторы объектов синхронизации ядра системы, которые видны во всей операционной системе и могут использоваться для синхронизации между процессами:

- Mutex.
- Semaphore (в Windows).
- EventWaitHandle (в Windows).

Облегченные типы синхронизации не соответствуют объектам ядра операционной системы и обычно обеспечивают лучшую производительность. Однако их нельзя использовать для синхронизации между процессами. Только для синхронизации потоков в одном приложении.

Некоторые из этих типов являются альтернативами типам, производным от WaitHandle. Например, SemaphoreSlim - это облегченная версия Semaphore.

Синхронизация доступа к общим ресурсам

Класс Monitor

Класс System.Threading.Monitor предоставляет исключительный доступ к общему ресурсу путем получения или снятия блокировки объекта, который идентифицирует ресурс. Пока блокировка удерживается, поток, удерживающий блокировку, может снова получить и освободить блокировку. Любой другой поток заблокирован для получения блокировки, и метод Monitor.Enter ждет, пока блокировка не будет снята. Метод Enter получает снятую блокировку. Можно также использовать метод Monitor.TryEnter, чтобы указать количество времени, в течение которого поток пытается получить блокировку. Поскольку класс Monitor имеет привязку к потокам, поток, получивший блокировку, должен снять блокировку, вызвав метод Monitor.Exit.

Можно координировать взаимодействие потоков, получающих блокировку использую методы Monitor.Wait, Monitor.Pulse, и Monitor.PulseAll.

Детали:
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.monitor

Класс Mutex

Класс System.Threading.Mutex, как и Monitor, предоставляет монопольный доступ к общему ресурсу. Для запроса владение мьютексом следует использовать одну из перегрузок метода Mutex.WaitOne. Как и Monitor, Mutex имеет привязку к потокам, и поток, получивший мьютекс, должен освободить его, вызвав метод Mutex.ReleaseMutex.

В отличие от монитора, мьютекс может использоваться для межпроцессной синхронизации. Для этого следует использовать именованный мьютекс, который виден во всей операционной системе. Для создания именованного экземпляра мьютекса, используются варианты конструктора, которые задают имя. Также можно вызвать метод Mutex.OpenExisting, чтобы открыть существующий именованный системный мьютекс.

Детали:
https://docs.microsoft.com/ru-ru/dotnet/standard/threading/mutexes
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.mutex

Структура SpinLock

Структура System.Threading.SpinLock, как и Monitor, предоставляет монопольный доступ к общему ресурсу в зависимости от доступности блокировки. Когда SpinLock пытается получить блокировку, которая недоступна, он ждет в цикле, многократно проверяя, пока блокировка не станет доступной. Но, при этом цикл загружает процессор. Т.о. использование SpinLock целесообразно, когда ожидается малое время ожидания.

Детали:
https://docs.microsoft.com/ru-ru/dotnet/standard/threading/spinlock
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.spinlock

Класс ReaderWriterLockSlim

Класс System.Threading.ReaderWriterLockSlim предоставляет эксклюзивный доступ к общему ресурсу для записи и позволяет нескольким потокам одновременно обращаться к ресурсу для чтения. Можно использовать ReaderWriterLockSlim для синхронизации доступа к общей структуре данных, которая поддерживает поточно-безопасные операции чтения, но требует монопольного доступа для записи. Когда поток запрашивает монопольный доступ (например, вызывая метод ReaderWriterLockSlim.EnterWriteLock), последующие запросы чтения и записи блокируются до тех пор, пока все существующие средства чтения не выйдут из блокировки, а средство записи не войдет в блокировку и не выйдет из нее.

Детали:
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.readerwriterlockslim

Классы Semaphore и SemaphoreSlim.

Классы System.Threading.Semaphore и System.Threading.SemaphoreSlim ограничивают количество потоков, которые могут одновременно обращаться к общему ресурсу или пулу ресурсов. Дополнительные потоки, запрашивающие ресурс, ждут, пока какой-либо поток не освободит семафор. Поскольку семафор не имеет привязки к потокам, один поток может получить семафор, а другой - освободить его.

SemaphoreSlim - это легкая альтернатива Semaphore, которую можно использовать только для синхронизации в пределах одного процесса.

В Windows можно использовать Semaphore для межпроцессной синхронизации. Для этого создается экземпляр Semaphore, представляющий именованный системный семафор, используя один из конструкторов Semaphore, который задает имя, или метод Semaphore.OpenExisting. SemaphoreSlim не поддерживает именованные системные семафоры.

Детали:
https://docs.microsoft.com/ru-ru/dotnet/standard/threading/semaphore-and-semaphoreslim
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.semaphoreslim

Взаимодействие потоков или сигнализирование

Взаимодействие потоков (или сигнализирование) означает, что поток должен ждать уведомления или сигнала от одного или нескольких потоков, чтобы продолжить. Например, если поток A вызывает метод Thread.Join потока B, поток A блокируется до завершения потока B. Примитивы синхронизации, описанные выше, предоставляют другой механизм для сигнализации: освобождая блокировку, поток уведомляет другой поток о том, что он может продолжить, установив блокировку.

Далее речь пойдет о дополнительных объектах сигнализирования, доступных в .NET.

Классы EventWaitHandle, AutoResetEvent, ManualResetEvent, and ManualResetEventSlim

Класс System.Threading.EventWaitHandle представляет событие синхронизации потоков.

Синхронизирующее событие может быть в одном из двух состояний: сигнальное и несигнальное. Когда событие находится в несигнальном состоянии поток, вызвавший одну из перегрузок метода WaitOne события блокируется до тех пор, пока событие не перейдет в сигнальное состояние.

Поведение при переходе в сигнальное состояние зависит от его режима сброса:

- EventWaitHandle, созданный с режимом EventResetMode.AutoReset сбрасывается автоматически после освобождения одного ожидающего потока. Это работает как турникет, который пропускает потоки по одному по сигналу. Класс System.Threading.AutoResetEvent, который наследуется от EventWaitHandle, реализует такой функционал.
- EventWaitHandle, созданный с режимом EventResetMode.ManualReset остаются в сигнальном состоянии до тех пор, пока не будет вызван метод Reset. Это работает как ворота, которые закрыты до поступления сигнала, а затем остаются открытыми до тех пор, пока кто-нибудь не закроет их. System.Threading.ManualResetEvent, который наследуется от EventWaitHandle, реализует такой функционал. Класс System.Threading.ManualResetEventSlim является облегченной версией ManualResetEvent.

В Windows можно использовать EventWaitHandle для синхронизации между процессами. Для этого необходимо создать экземпляр EventWaitHandle, представляющий именованное системное событие синхронизации используя один из конструкторов EventWaitHandle, которые задают имя. Либо используя метод EventWaitHandle.OpenExisting получить доступ к существующему событию.

Детали:
https://docs.microsoft.com/ru-ru/dotnet/standard/threading/eventwaithandle
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.eventwaithandle
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.autoresetevent
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.manualresetevent
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.manualreseteventslim

Класс CountdownEvent

Класс System.Threading.CountdownEvent представляет собой событие срабатывает, когда его счетчик становится равным нулю. Пока CountdownEvent.CurrentCount больше нуля, поток, вызывающий CountdownEvent.Wait блокируется. Вызов CountdownEvent.Signal уменьшает на единицу счетчик события.

В отличии от ManualResetEvent или ManualResetEventSlim, которые можно использовать для разблокировки нескольких потоков по сигналу от одного потока, CountdownEvent разблокирует один или несколько потоков по сигналам от нескольких потоков.

Детали:
https://docs.microsoft.com/ru-ru/dotnet/standard/threading/countdownevent
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.countdownevent

Класс Barrier

Класс System.Threading.Barrier представляет собой барьер выполнения потока. Поток, вызывающий метод Barrier.SignalAndWait сигнализирует, что он достиг барьера и ожидает пока остальные участвующие потоки достигнут барьера. Когда все участвующие потоки достигают барьера, они продолжаются и барьер сбрасывается и может использоваться повторно.

Barrier можно использовать, когда один и более потоков нуждаются в результатах других потоков перед переходом к следующему этапу вычислений.

Детали:
https://docs.microsoft.com/ru-ru/dotnet/standard/threading/barrier
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.barrier

Класс Interlocked

Класс System.Threading.Interlocked предоставляет статические методы для выполнения атомарных операций над переменной. Атомарные операции включают в себя: сложение, инкремент и декремент, обмен, условный обмен, чтение.

Детали:
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.interlocked

Структура SpinWait

Структура System.Threading.SpinWait предоставляет поддержку ожидающего цикла. Она предназначена для использования ожидания события или выполнения условия, но когда ожидается, что время ожидания меньше, чем при использовании EventWaitHandle или другого примитива синхронизации (вызов ядра, переключение контекста процесса или потока – достаточно длительные операции). При использовании SpinWait можно указать период времени выполнения цикла ожидания перед покиданием процессора (ожиданием или сном) если ожидаемое условие не выполнилось.

Детали:
https://docs.microsoft.com/ru-ru/dotnet/standard/threading/spinwait
https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.spinwait

Индивидуальные задания

Необходимо разработать однопоточное и многопоточное приложение, решающих задачу согласно варианта с использованием указанного способа реализации многопоточности и механизма синхронизации.

Произвести замеры времени высокоточным таймером из пространства имен System.Diagnostics. Сравнить полученные результаты при различной размерности решаемых задач.

Задания разместить в файле «tasks.txt», где перечислить в строках список заданий. Например:
a1.csv,b1.csv
a2.csv,b2.csv
…
В файлах «a1.csv», «b1.csv» разместить матрицы коэффициентов уравнений и векторы правых частей. Количество задач – не менее 10-ти, размерность задач – не менее 100.

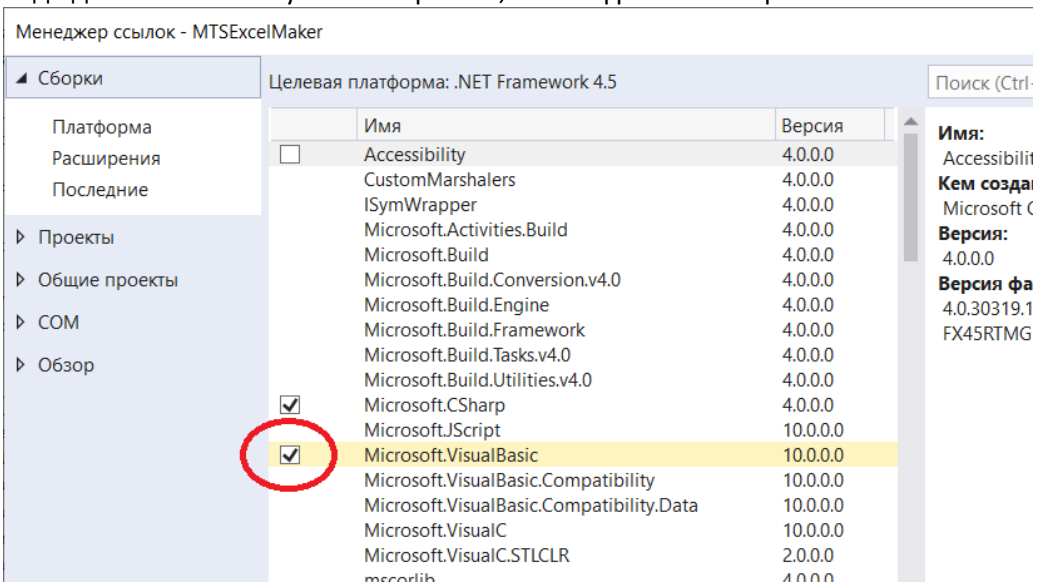
Подсказка

Наборы исходных данных можно сгенерировать случайно в MathCAD: матрицу A (причем, для способов разложения можно сгенерировать L (и U, или D), затем из них получить A) и вектор решений x, затем, перемножив, получить вектор B.

Затем сохранить в файлах формата CSV (Comma Separated Values) используя функцию «WRITECSV», как показано на фрагменте скриншота (показана версия 15, но аналогичные функции присутствуют и в более поздних версиях):

$$tmp := WRITECSV(values, "a1.csv")$$

Импорт можно осуществить стандартными средствами платформы .NET. Однако, это не так очевидно. Парсер CSV находится в сборке «Microsoft.VisualBasic». Ее надо добавить к списку ссылок проекта, как на фрагменте скриншота:



После добавления ссылки, добавив
```
using Microsoft.VisualBasic.FileIO;
```
можно воспользоваться классом
```
TextFieldParser csvParser = new TextFieldParser(fileName);
```

Поскольку MathCAD (по крайней мере версии до 15-ой включительно) используют преобразование числа в строку инвариантное к региональным настройкам (разделитель целой и дробной частей - точка), при разборе файлов необходимо использовать перегрузки методов с заданием региональных настроек, подставив инвариантную:
```
using System.Globalization;
…
   CultureInfo invariantCulture = CultureInfo.InvariantCulture;
…
      … = double.Parse( … , invariantCulture);
```

После разбора всех заданий запустить процесс решения с замером времени. Количество потоков должно быть равно количеству ядер процессора, а при использовании пула потоков необходимо ограничить максимальное значение потоков числом ядер + 4. По завершении решения результаты (значения корней) записать в файлы «x1.csv», «x2.csv», … Результаты замера времени вывести в консоль.

№ варианта	Алгоритм, исполняемый процессами	Способ реализации многопоточности	Механизм синхронизации
1	Решение СЛАУ методом Гаусса	Потоки	Атомарные операции
2	Решение СЛАУ методом Зейделя	Пул потоков	Семафор
3	Решение СЛАУ методом Гаусса-Зейделя	Потоки	Критическая секция
4	Решение СЛАУ методом LL^T разложения	Пул потоков	Мютекс
5	Решение СЛАУ методом LU разложения	Потоки	Событие
6	Решение СЛАУ методом LDL^T разложения	Пул потоков	Семафор
7	Решение СЛАУ методом Гаусса	Потоки	Атомарные операции
8	Решение СЛАУ методом Зейделя	Пул потоков	Семафор
9	Решение СЛАУ методом Гаусса-Зейделя	Потоки	Критическая секция
10	Решение СЛАУ методом LL^T разложения	Пул потоков	Мютекс
11	Решение СЛАУ методом LU разложения	Потоки	Событие
12	Решение СЛАУ методом LDL^T разложения	Пул потоков	Семафор
13	Решение СЛАУ методом LL^T разложения	Потоки	Семафор
14	Решение СЛАУ методом LDL^T разложения	Пул потоков	Критическая секция
15	Решение СЛАУ методом Гаусса	Потоки	Критическая секция