Структура SpinLock

В .NET 4 появилась структура SpinLock. Она может применяться в случае, когда накладные расходы, связанные с объектами блокировки (Monitor), оказываются слишком высокими из-за сборки мусора. Эта структура особенно полезна при большом количестве блокировок (например, для каждого узла в списке) и чрезмерно коротких периодах их удержания. Следует стараться избегать использования более чем одной структуры SpinLock и не вызывать ничего, что может приводить к блокировке.

За исключением различий в архитектуре, структура SpinLock в плане применения очень похожа на [класс Monitor](https://professorweb.ru/my/csharp/thread_and_files/1/1_9.php). Получение блокировки осуществляется с помощью методов Enter() или TryEnter(), а снятие — с помощью метода Exit(). Кроме того, SpinLock предлагает свойства для предоставления информации в случае, если в текущий момент находится в заблокированном состоянии: IsHeld и IsHeldByCurrentThread.

При передаче экземпляров SpinLock следует соблюдать осторожность. Из-за того, что SpinLock определена как struct, ее присваивание приводит к созданию копии. Поэтому экземпляры SpinLock должны всегда передаваться по ссылке.

Класс Barrier

Для синхронизации в .NET 4 предлагается новый класс Barrier. Этот класс прекрасно подходит для сценариев, когда работа разбивается на множество задач и должна позже снова объединяться в одно целое. Он применяется для участников, нуждающихся в синхронизации. До тех пор, пока задание остается активным, динамически могут добавляться дополнительные участники, например, дочерние задачи, создаваемые из родительской задачи. Эти участники могут ожидать, пока все остальные участники не выполнят свою работу.

В предлагаемом здесь для примера приложении используются коллекция, содержащая 2.000.000 строк. Множество задач производят проход по этой коллекции с подсчетом количества строк, начинающихся с "а", "Ь", "с" и т.д.

Метод FillData() создает коллекцию и заполняет ее произвольными строками:

using System;

using System.Threading;

using System.Threading.Tasks;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

public static IEnumerable<string> FillData(int size)

{

List<string> data = new List<string>(size);

Random r = new Random();

for (int i = 0; i < size; i++)

data.Add(GetString(r));

return data;

}

private static string GetString(Random r)

{

StringBuilder sb = new StringBuilder(6);

for (int i = 0; i < 6; i++)

sb.Append((char)(r.Next(26) + 97));

return sb.ToString();

}

...

Метод CalculationlnTask() определяет задание, которое должно выполняться в рамках задачи. В параметре передается кортеж, содержащий четыре элемента. Третий элемент кортежа представляет собой ссылку на экземпляр Barrier. После выполнения задания задача удаляет себя из Barrier с помощью метода RemoveParticipant():

static int[] CalculationInTask(object p)

{

var p1 = p as Tuple<int, int, Barrier, List<string>>;

Barrier barrier = p1.Item3;

List<string> data = p1.Item4;

int start = p1.Item1 \* p1.Item2;

int end = start + p1.Item2;

Console.WriteLine("Задача {0}: раздел от {1} до {2}",

Task.CurrentId,start,end);

int[] charCount = new int[26];

for (int j = start; j < end; j++)

{

char c = data[j][0];

charCount[c - 97]++;

}

Console.WriteLine("Задача {0} завершила вычисление. {1} раз а, {2} раз z",

Task.CurrentId, charCount[0], charCount[25]);

barrier.RemoveParticipant();

Console.WriteLine("Задача {0} удалена; количество оставшихся участников: {1}",

Task.CurrentId,barrier.ParticipantsRemaining);

return charCount;

}

Экземпляр Barrier создается в методе Main(). В его конструкторе можно указывать желаемое количество участников. В рассматриваемом здесь примере это количество равно 3, поскольку создаются две задачи, плюс сам метод Main() тоже является участником. Эти две задачи создаются с помощью TaskFactory, обеспечивая разбиение процесса прохода по коллекции на две части. После запуска этих задач с помощью метода SignalAndWait() метод Main() сигнализирует о своем завершении и ожидает, пока все остальные участники либо тоже просигнализируют о своем завершении, либо удалят себя как участников из Barrier. Как только все участники готовы, все возвращенные ими результаты объединяются вместе с помощью метода расширения Zip():

static void Main()

{

const int numberTasks = 2;

const int partitionSize = 1000000;

var data = new List<string>(FillData(partitionSize \* numberTasks));

var barrier = new Barrier(numberTasks + 1);

var taskFactory = new TaskFactory();

var tasks = new Task<int[]>[numberTasks];

for (int i = 0; i < numberTasks; i++)

{

tasks[i] = taskFactory.StartNew<int[]>(CalculationInTask,

Tuple.Create(i, partitionSize, barrier, data));

}

barrier.SignalAndWait();

var resultCollection = tasks[0].Result.Zip(tasks[1].Result, (c1, c2) =>

{

return c1 + c2;

});

char ch = 'a';

int sum = 0;

}

ReaderWriterLockSlim

https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.readerwriterlockslim?view=net-5.0