**Параллельное программирование**

Параллельное программирование — это метод выполнения нескольких вычислительных задач одновременно, что позволяет ускорить обработку данных и улучшить производительность приложений. В контексте системного программирования на C# это особенно полезно для работы с процессами, потоками и тяжелыми вычислениями.

Основные цели параллельного программирования:

Ускорение выполнения задач. Распараллеливание задач позволяет обрабатывать их быстрее.

Улучшение отзывчивости. При параллельной обработке системные приложения могут быть более отзывчивыми, особенно при работе с ресурсами, которые занимают много времени.

Оптимизация использования ресурсов. Параллельное программирование позволяет лучше использовать доступные ядра процессора.

Библиотека параллельных задач (TPL) в C#

В C# для параллельного программирования чаще всего используется библиотека System.Threading.Tasks, которая включает классы Task, Parallel и другие.

(Пример 1): Использование Task для параллельного выполнения задач

Класс Task позволяет создавать и запускать задачи, которые будут выполняться параллельно.

В этом примере задачи task1 и task2 выполняются параллельно, а программа ожидает их завершения с помощью Task.WaitAll.

(Пример 2): Использование Parallel.For для параллельных циклов

Parallel.For позволяет выполнять итерации цикла одновременно, что значительно сокращает время выполнения.

Здесь каждая итерация цикла Parallel.For запускается параллельно, что может быть полезно для выполнения тяжелых операций в каждой итерации.

Параллельное программирование особенно полезно в системном программировании для управления потоками, обработки больших объемов данных или выполнения операций, требующих высокой вычислительной мощности.

**Классы Task and Parallel в системном программировании**

Класс Task представляет собой асинхронную операцию и предназначен для организации параллельных вычислений. Основные задачи Task — это управление выполнением задач, получение их результатов и обработка исключений, возникающих при выполнении.

Цели и задачи класса Task

Асинхронность. Выполнение задач в отдельном потоке, чтобы освободить основной поток.

Параллельное выполнение. Ускорение программы за счет распараллеливания вычислений.

Управление состоянием задачи. Позволяет следить за выполнением, завершением или отменой задач.

Методы класса Task

Run(Action) — запускает задачу в новом потоке и возвращает объект Task.

Start() — начинает выполнение задачи, которая была создана, но еще не запущена.

Wait() — приостанавливает выполнение вызывающего потока до завершения задачи.

ContinueWith() — задает задачу, которая будет выполнена после завершения текущей задачи.

Result — возвращает результат выполнения задачи (если задача возвращает значение).

Свойства класса Task

IsCompleted — указывает, завершена ли задача.

Status — возвращает текущее состояние задачи (WaitingToRun, Running, Completed и др.).

Exception — возвращает исключения, возникшие в процессе выполнения задачи.

(Пример 3) - В этом примере создается задача, которая вычисляет сумму чисел, а результат используется после завершения задачи.

**Класс Parallel в C#**

Класс Parallel предоставляет методы для выполнения циклов и вызова действий параллельно. Цель Parallel — распараллеливать операции, которые могут выполняться одновременно, и улучшить производительность.

Цели и задачи класса Parallel

Параллельное выполнение циклов. Выполнение итераций цикла одновременно.

Упрощение синтаксиса параллелизма. Предоставление методов для удобного параллельного вызова действий.

Методы класса Parallel

For(int fromInclusive, int toExclusive, Action<int>) — выполняет параллельно итерации цикла.

ForEach(IEnumerable<TSource>, Action<TSource>) — выполняет параллельно действия для каждого элемента коллекции.

Invoke(params Action[]) — позволяет выполнить несколько действий параллельно.

(Пример 4) - Здесь Parallel.For выполняет итерации одновременно, а Parallel.ForEach выполняет действия для каждого элемента коллекции параллельно.

**Структуры данных для параллельного программирования**

В параллельном программировании в C# используются специальные структуры данных, которые обеспечивают безопасный доступ из нескольких потоков одновременно. Эти структуры разработаны так, чтобы избегать условий гонки и обеспечивать высокую производительность при параллельных операциях.

Основные структуры данных для параллельного программирования

**ConcurrentBag<T>**

Представляет собой неблокирующий, потокобезопасный контейнер для хранения объектов в произвольном порядке.

Хорошо подходит для ситуаций, когда порядок объектов не важен и доступ к ним может быть параллельным.

Поддерживает несколько потоков для добавления и удаления элементов без блокировки. (пример 5).

**ConcurrentQueue<T>**

Потокобезопасная очередь, поддерживающая доступ из нескольких потоков без блокировки.

Элементы добавляются в конец и извлекаются из начала, что делает очередь полезной для задач, работающих по принципу FIFO (First-In-First-Out).

Часто используется для передачи задач между потоками. (пример 6).

**ConcurrentStack<T>**

Потокобезопасный стек, поддерживающий доступ из нескольких потоков без блокировки.

Работает по принципу LIFO (Last-In-First-Out), поэтому новые элементы добавляются и удаляются с верхушки стека. (пример 7).

**ConcurrentDictionary<TKey, TValue>**

Потокобезопасный словарь, обеспечивающий безопасное добавление, удаление и изменение элементов.

Хорошо подходит для хранения и доступа к данным в многопоточной среде, где требуется доступ по ключу.

Поддерживает неблокирующие методы добавления и обновления значений. (пример 8).

**BlockingCollection<T>**

Коллекция, которая поддерживает блокировку при добавлении или удалении элементов.

Часто используется для передачи данных между потоками с возможностью блокировки при достижении предельного количества элементов.

Поддерживает различные внутренние коллекции, такие как ConcurrentQueue или ConcurrentStack, и позволяет гибко выбирать модель добавления и извлечения данных. (пример 9).

**Parallel LINQ (PLINQ)**

PLINQ (Parallel LINQ) — это расширение для LINQ в C#, которое позволяет выполнять запросы к коллекциям параллельно, используя несколько потоков. PLINQ автоматически распределяет работу между доступными процессорами, что помогает ускорить обработку больших объемов данных.

**Отличия LINQ от PLINQ**

Параллелизм: PLINQ выполняет запросы параллельно, тогда как LINQ работает в одном потоке.

Производительность: PLINQ может улучшить скорость обработки данных на многопроцессорных системах.

Управление порядком: В PLINQ порядок элементов может нарушиться, хотя есть параметры для его сохранения.

Обработка исключений: PLINQ объединяет исключения из всех потоков, тогда как LINQ выбрасывает их сразу.

**Класс ParallelEnumerable**

Класс ParallelEnumerable — это основа PLINQ и предоставляет методы для создания параллельных запросов.

**Цели и задачи ParallelEnumerable**

Оптимизация запросов к коллекциям путем распределения нагрузки по потокам.

Сохранение простоты LINQ-синтаксиса при параллельной обработке данных.

Управление параллельным выполнением и настройка поведения параллельного запроса.

**Методы класса ParallelEnumerable**

AsParallel() — преобразует последовательность в параллельную последовательность.

AsOrdered() — сохраняет порядок элементов в параллельной последовательности.

WithDegreeOfParallelism(int) — устанавливает число потоков для выполнения запроса.

WithExecutionMode(ParallelExecutionMode) — задает режим выполнения (параллельный или автоматический).

ForAll(Action<T>) — выполняет действие для каждого элемента параллельной последовательности. (пример 10)

**Скорость выполнения в PLINQ**

PLINQ часто быстрее LINQ на больших объемах данных, особенно на многопроцессорных системах, так как запросы выполняются параллельно. Однако скорость может зависеть от типа операций и сложности данных.

**Факторы, влияющие на скорость выполнения**

Объем данных: чем больше данных, тем больше прирост производительности.

Тип операций: операции с минимальными задержками (например, фильтрация) быстрее в PLINQ.

Число потоков: при правильной настройке потоков (с помощью WithDegreeOfParallelism) можно увеличить производительность.

Операции ввода-вывода: такие операции лучше оставить для последовательного выполнения, так как они могут замедлить параллельные запросы. (пример 11).

**Пример с сохранением порядка (AsOrdered)**

Предположим, у нас есть коллекция чисел, и мы хотим параллельно удвоить значения, но при этом сохранить изначальный порядок элементов. Если использовать AsParallel без AsOrdered, порядок элементов может измениться, что может быть критичным для некоторых задач. (пример 12).

**Пример с параметрами для слияния (WithMergeOptions)**

В некоторых случаях нам нужно контролировать, как результаты параллельной обработки собираются. Например, ParallelMergeOptions.NotBuffered позволяет использовать результаты по мере их готовности, а не дожидаться окончания всех операций. Это может быть полезно для снижения задержки при обработке данных, особенно если часть данных может быть обработана раньше.

В этом примере мы используем ParallelMergeOptions.NotBuffered, чтобы сразу обрабатывать готовые результаты и видеть их в консоли. (пример 13)