1. Что такое *TPL*? Как и для чего используется тип *Task*

Библиотека параллельных задач, позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах (для создания многопоточных приложений)

**Task описывает отдельную продолжительную операцию, которая запускается асинхронно в одном из потоков из пула потоков (можно запускать синхронно в текущем потоке) – подобна потокам, но абстракция более высокого уровня**

1. Почему эффект от распараллеливания наблюдается на большом количестве элементов?

Эффект от распараллеливания наиболее заметен на большом объеме данных, так как распараллеливание позволяет нескольким потокам одновременно работать над разными частями задачи, ускоряя выполнение. На маленьких объемах данных накладные расходы на создание потоков и управление ими могут превышать выгоду от распараллеливания. Однако, с увеличением объема работы эти накладные расходы становятся менее значимыми, и распараллеливание начинает приносить ощутимое ускорение. Когда данных много, каждый поток выполняет свою часть задачи, и время работы программы значительно сокращается.

3. В чем основные достоинства работы с задачами по сравнению с потокми?

Задачи (Task) предлагают более высокий уровень абстракции для работы с многопоточностью, чем потоки (Thread).

Task управляет внутренним потоком автоматически

задачи, созданные через Task, по умолчанию используют механизм пула потоков. Это позволяет эффективно управлять потоками, обеспечивая высокую производительность и экономию ресурсов, а разработчику не нужно вручную создавать или уничтожать потоки.

4. Приведите три способа создания и/или запуска Task?

* Первый способ создание объекта Task и вызов у него метода Start:



Второй способ заключается в использовании статического метода Task.Factory.StartNew



* Третий способ определения и запуска задач представляет использование статического метода Task.Run():



5. Как и для чего используют методы *Wait*(), *WaitAll*() и *WaitAny*()?

**Wait():**

* **Что делает:** Блокирует текущий поток до завершения выполнения задачи.

**WaitAll():**

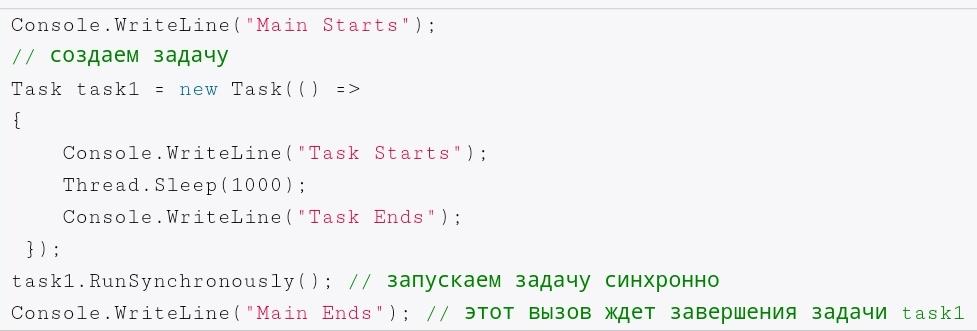
* **Что делает:** Блокирует текущий поток до завершения всех указанных задач.

**WaitAny():**

* **Что делает:** Блокирует текущий поток до завершения хотя бы одной из указанных задач.

6. Приведите пример синхронного запуска *Task*?

* По умолчанию задачи запускаются асинхронно
* RunSynchronously() - можно запускать синхронно



7. Как создать задачу с возвратом результата?

Task<TResult> - описывает задачу, возвращающую значение типа Tresult

8. Как обработать исключение, если оно произошло при выполнении *Task*?

9. Что такое *CancellationToken* и как с его помощью отменить выполнение задач?

* Структура CancellationToken - токен отмены
* Создание объекта CancellationTokenSource, который управляет и посылает уведомление об отмене токену.
* С помощью свойства CancellationTokenSource.Token получаем собственно токен - объект структуры CancellationToken и передаем его в задачу, которая может быть отменена.
* Для передачи токена в задачу можно применять один из конструкторов класса Task

Определяем в задаче действия на случай ее отмены.

4. Вызываем метод CancellationTokenSource.Cancel(), который устанавливает для свойства CancellationToken.IsCancellationRequested значение true.

* метод CancellationTokenSource.Cancel() не отменяет задачу, он лишь посылает уведомление об отмене

Каким образом будет происходить выход из задачи, это решает сам разработчик.

5. Класс **CancellationTokenSource** реализует интерфейс IDisposable. И когда работа с объектом CancellationTokenSource завершена, у него следует вызвать метод Dispose для освобождения всех связанных с ним используемых ресурсов.

10. Как организовать задачу продолжения (continuation task) ?

Задача продолжения (continuation task) — это задача, которая запускается после завершения другой задачи. Она позволяет создавать цепочки задач, чтобы упорядочить их выполнение.

Для создания задачи продолжения в .NET используется метод ContinueWith. Этот метод позволяет связать задачу с другой задачей, которая будет выполнена, когда первая завершится.

11. Как и для чего используется объект ожидания при создании задач продолжения?

* Объект ожидания – это любой объект, имеющий методы OnCompleted() и GetResult() и свойство IsCompleted.

Объект ожидания (например, метод Wait или TaskCompletionSource) используется для синхронизации выполнения программы с завершением задач или цепочки задач. Это необходимо, чтобы гарантировать, что задачи завершены до выполнения других операций.

12. Поясните назначение класса *System.Threading.Tasks.Parallel*?

Класс System.Threading.Tasks.Parallel предоставляет методы для выполнения циклов (for и foreach) и других операций параллельно. Он используется для распараллеливания вычислительных задач, которые могут быть разделены на несколько независимых частей.

13. Приведите пример задачи с *Parallel.For(int, int, Action<int>)*

тот код выполняет задачу параллельного вычисления квадратов чисел от 1 до 10 с использованием класса Parallel.  Каждая итерация цикла (вычисление квадрата числа) выполняется параллельно в разных потоках, что позволяет ускорить выполнение задачи при большом количестве итераций.

 **Порядок выполнения**: Поскольку Parallel.For выполняет итерации параллельно, порядок вывода чисел на экран может не совпадать с порядком чисел от 1 до 10. То есть, строки могут выводиться в произвольном порядке,

14. Приведите пример задачи с *Parallel.ForEach*

Метод Parallel.ForEach() используется для параллельного выполнения задач, каждая из которых работает с элементами коллекции (например, массив, список, словарь и т.д.). Этот метод позволяет обрабатывать все элементы коллекции одновременно, используя несколько потоков, что может значительно ускорить выполнение программы, если количество элементов в коллекции велико и операции с ними независимы.

15. Приведите пример с *Parallel.Invoke()*

Метод Parallel.Invoke() используется для параллельного выполнения нескольких независимых задач. Он позволяет запустить несколько действий (методов или лямбда-выражений) одновременно, не требуя явного управления потоками. Каждое действие выполняется в отдельном потоке, и выполнение этих действий происходит параллельно

16. Как с использованием *CancellationToken* отменить параллельные операции?

1. Создание объекта CancellationTokenSource, который управляет и посылает уведомление об отмене токену.
2. С помощью свойства CancellationTokenSource.Token получаем собственно токен - объект структуры CancellationToken и передаем его в задачу, которая может быть отменена.

3. Определяем в задаче действия на случай ее отмены.

4. Вызываем метод CancellationTokenSource.Cancel(), который устанавливает для свойства CancellationToken.IsCancellationRequested значение true.

* метод CancellationTokenSource.Cancel() не отменяет задачу, он лишь посылает уведомление об отмене

5. Класс **CancellationTokenSource** реализует интерфейс IDisposable. И когда работа с объектом CancellationTokenSource завершена, у него следует вызвать метод Dispose для освобождения всех связанных с ним используемых ресурсов.

17. Для чего используют *BlockingCollection*<T>, в чем ее особенность?

BlockingCollection<T> — это потокобезопасная коллекция, предназначенная для организации обмена данными между потоками (продуцентами и потребителями).

#### Особенности:

* Поддерживает ограничение на максимальное количество элементов (буферизация).
* Потоки автоматически блокируются, если:
  + Коллекция пуста (потребитель ждёт данных).
  + Коллекция переполнена (продуцент ждёт места).
* Обеспечивает потокобезопасность без необходимости использования lock.

18. Как используя *async* и *await* организовать асинхронное выполенение метода?

**Определите метод как async**:

* Пометьте метод, который будет выполняться асинхронно, ключевым словом async. Это позволяет использовать await внутри метода.

Асинхронные методы должны возвращать Task

**Используйте await для асинхронных операций**:

* Внутри метода используйте await перед вызовом асинхронного метода. Это позволяет дождаться завершения операции без блокировки потока. await позволяет приостановить выполнение метода до тех пор, пока не завершится асинхронная операция

вызов

using System;

using System.Collections.Generic;

using System.Collections.Specialized;

using System.Data;

using System.Diagnostics;

using System.Collections.Concurrent;

namespace Lab\_15\_OOP

{

public class Program

{

public static void Main()

{

//// Создаем источник отмены

///Источник отмены — это специальный объект в .NET, который помогает отменять задачи или операции.

// Его можно представить как пульт управления, с помощью которого вы можете сказать: "Стоп, хватит!" всем задачам, связанным с этим пультом.Когда вы нажимаете кнопку отмены(вызываете метод Cancel()), задачи получают сигнал и могут завершить свою работу.

CancellationTokenSource cancellationTokenSource = new CancellationTokenSource(); //создается объект cancellationTokenSource, который предоставляет возможность управлять отменой операций. Этот объект создается, чтобы контролировать отмену задачи.

// Он предоставляет метод Cancel(), который сообщает связанным токенам, что операция должна быть отменена.

///////////////////////Создание объекта CancellationTokenSource, который управляет и посылает уведомление об отмене токену.

//////////////////////////////С помощью свойства CancellationTokenSource.Token получаем собственно токен - объект структуры CancellationToken и передаем его в задачу, которая может быть отменена.

//токен Передает сигнал об отмене от управляющего компонента (например, CancellationTokenSource)

CancellationToken tok = cancellationTokenSource.Token; //Это токен, связанный с объектом CancellationTokenSource. Он:

// Передается в задачи или методы, чтобы они могли отслеживать, была ли инициирована отмена.

//Содержит свойство IsCancellationRequested, которое возвращает true, если была вызвана отмена через CancellationTokenSource.Cancel().

bool[] resheto = new bool[100001];//Создается массив, где true означает, что число может быть простым.

//Используется метод "Решето Эратосфена", чтобы найти все простые числа до заданного значения n.

for (int i = 0; i < 100001; i++) // цикл перебирает все индексы массива от 0 до 100000 и устанавливает каждому элементу значение true.

{

resheto[i] = true;

}

List<int> primeNumbers = new List<int>();//Это список, в который добавляются найденные простые числа.

void searchPrimeNum(int n) //Алгоритм реализует "Решето Эратосфена", чтобы найти все простые числа до заданного числа n.

{

if (tok.IsCancellationRequested) //Если был запрос на отмену задачи (CancellationToken), выполнение прерывается.

{

Console.WriteLine("Операция была прервана");

return;

}

for (int i = 2; i <= n; i++)

{

if (resheto[i]) //Перебираются числа от 2 до n (т.к. 0 и 1 не являются простыми числами).

// Если текущий элемент resheto[i] равен true, это число считается простым и добавляется в список primeNumbers.

{

primeNumbers.Add(i);

}

for (int j = i \* i; j <= n; j += i) //Когда найдено простое число i, все его кратные помечаются как составные (false в массиве resheto).

//Цикл начинается с i \*i, потому что все числа меньше i \*i уже были обработаны ранее.

{

resheto[j] = false;

}

}

}

int n = int.Parse(Console.ReadLine()); //Запрашивается ввод числа n с клавиатуры

Task searchPrimeNumbers = new Task(() => searchPrimeNum(n)); //Создаётся задача searchPrimeNumbers, которая будет выполнять функцию searchPrimeNum(n) — поиск простых чисел до n.

// Задача пока только создана, но ещё не запущена.

Console.WriteLine($"Идентификатор задачи: {searchPrimeNumbers.Id}");

Console.WriteLine($"Статус задачи до начала выполнения: {searchPrimeNumbers.Status}");

Stopwatch stopwatch = new Stopwatch(); //Предоставляет набор методов и свойств, которые можно использовать для точного измерения затраченного времени.

stopwatch.Start(); //Запускается таймер.

searchPrimeNumbers.Start(); //Задача searchPrimeNumbers начинает выполняться. Теперь функция searchPrimeNum(n) ищет простые числа.

Console.WriteLine($"Статус задачи во время выполнения: {searchPrimeNumbers.Status}"); //Показывает текущий статус задачи (например, Running — задача выполняется).

searchPrimeNumbers.Wait(); //Программа ждёт, пока задача завершит выполнение.

stopwatch.Stop(); //Таймер (stopwatch.Stop()) останавливается после завершения задачи.

Console.WriteLine($"Затраченное время: {stopwatch.Elapsed.TotalMilliseconds}"); //Печатает, сколько времени ушло на выполнение задачи в миллисекундах.

Console.WriteLine($"Статус задачи по завершению выполнения: {searchPrimeNumbers.Status}"); //Показывает финальный статус задачи (например, RanToCompletion — задача успешно завершена).

foreach (int i in primeNumbers) //Цикл берёт каждый элемент из этого списка по очереди и присваивает его переменной i

{

Console.Write($"{i} ");

}

Console.WriteLine();

primeNumbers.Clear();//Эта строка очищает список primeNumbers, удаляя все его элементы.;

///////////////////////////////////////////////////////////////////////////////

Task task2 = new Task(() => searchPrimeNum(n), tok); //Создаётся новая задача (Task), которая запускает метод searchPrimeNum(n). Этот метод ищет простые числа до заданного числа n.

// В задачу передаётся токен отмены(tok).Это значит, что задача может проверять, была ли отмена, и прерывать свою работу, если нужно.

//Почему используется токен?

//Чтобы можно было отменить выполнение задачи в любой момент с помощью метода cancellationTokenSource.Cancel().

task2.Start(); //адача task2 запускается в отдельном потоке и начинает выполнять метод searchPrimeNum(n).

cancellationTokenSource.Cancel(); //ызывается метод Cancel() у объекта cancellationTokenSource. Это отправляет сигнал отмены всем задачам, которые используют токен tok.

Thread.Sleep(200); //Программа приостанавливается на 200 миллисекунд (0,2 секунды), чтобы дать задаче время на выполнение и обработку сигнала отмены.

Console.WriteLine($"Статус задачи task2: {task2.Status}");

primeNumbers.Clear(); //список primeNumbers очищается

/////////////////////////////////////////////////////////////////////////////

int sum(int a, int b)

{

return a + b;

}

int mul(int a, int b)

{

return a \* b;

}

int sub(int a, int b)

{

return a - b;

}

//это задача (или операция), которая выполняется в фоновом потоке и в итоге вернет значение типа int.

Task<int> t1 = Task.Run(() => sum(1, 2));//Task.Run() — это метод, который позволяет запустить асинхронную операцию в отдельном потоке.

Task<int> t2 = Task.Run(() => mul(1, 2));

Task<int> t3 = Task.Run(() => sub(1, 2));

//WhenAll сначала завершатся все задачи, и лишь только потом будет выполняться последующий код

Task t4 = Task.WhenAll(t1, t2, t3).ContinueWith(t => //.ContinueWith() — это метод, который позволяет указать код, который нужно выполнить после завершения задачи

{

Console.WriteLine($"Сумма, произведение и разность чисел 1 и 2 соответственно равны {t1.Result}, {t2.Result}, {t3.Result}");

});

t4.Wait(); // говорит программе дождаться завершения задачи t4.

Task<int> t5 = Task.Run(() => sum(1, 2));

Task<int> t6 = Task.Run(() => mul(1, 2));

Task<int> t7 = Task.Run(() => sub(1, 2));

var awaiter1 = t5.GetAwaiter();//Метод GetAwaiter() возвращает специальный объект, который называется awaiter. Он будет отслеживать, когда задача t5 завершится.

//С помощью объекта awaiter1 мы можем настроить действия, которые произойдут, когда задача завершится.

awaiter1.OnCompleted(() => //Когда задача t5 завершится, сработает код внутри OnCompleted, и будет выведено сообщение о результате задачи

{

//Действие, которое необходимо выполнить после завершения операции ожидания.

Console.WriteLine($"Результат выполнения первой задачи {awaiter1.GetResult()}");

});

Task.WaitAll(t5, t6, t7); //программа приостанавливает выполнение до тех пор, пока все переданные задачи не завершатся. Это позволяет дождаться, когда все вычисления завершатся и мы получим результаты.

Console.WriteLine($"Сумма, произведение и разность чисел 1 и 2 соответственно равны {t5.Result}, {t6.Result}, {t7.Result}");

///////////////////////////////////////////////////////////////////////////////////

int[] arr1 = new int[1000000]; //два массива arr1 и arr2, каждый из которых может хранить миллион целых чисел.

int[] arr2 = new int[1000000];

void createMas(int i)

{

Random rand = new Random();//это метод, который принимает индекс массива и заполняет элемент в массиве arr1 случайным числом от 0 до 999.

arr1[i] = rand.Next(0, 1000);

}

void createMas2(int i)

{

Random rand = new Random();//работает аналогично, но для массива arr2.

arr2[i] = rand.Next(0, 1000);

}

Stopwatch timer = new Stopwatch(); //Stopwatch используется для измерения времени, которое затрачивается на выполнение этого параллельного цикла.

timer.Start();

Parallel.For(0, 1000000, createMas);//Parallel.For — это метод, который выполняет цикл с параллельным запуском. Вместо того чтобы выполнять каждую итерацию цикла последовательно, как это происходит в обычном for-цикле, Parallel.For разделяет работу между несколькими потоками, что ускоряет выполнение,

//Эти два числа — это пределы для выполнения цикла. Первый параметр (0) — это начальный индекс, с которого будет начинаться выполнение цикла, а второй параметр (1000000) — это конечный индекс, до которого будет идти цикл

// createMas

//Это метод, который будет вызываться на каждой итерации цикла

//Этот метод генерирует случайное число от 0 до 999 и записывает его в массив

timer.Stop();

Console.WriteLine($"Время выполнения Parallel.For: {timer.Elapsed.TotalMilliseconds}");//Время выполнения выводится в миллисекундах с помощью

Stopwatch timer2 = new Stopwatch();

timer2.Start();

for (int i = 0; i < 1000000; i++) //Это обычный цикл for, который будет выполняться миллион раз.

{

createMas2(i);

}

timer2.Stop();

Console.WriteLine($"Время выполнения For: {timer2.Elapsed.TotalMilliseconds}");

void createMas3(int i)

{

i += 5;//принимает один параметр типа int — индекс массива. Внутри метода выполняется операция, которая увеличивает значение этого индекса на 5

}

Stopwatch timer3 = new Stopwatch();

timer3.Start();

Parallel.ForEach(arr1, createMas3); // метод createMas3 будет параллельно вызываться для каждого элемента массива arr1 одновременно в разных потоках. Таким образом, обработка массива происходит гораздо быстрее по сравнению с обычным циклом for, потому что каждый элемент обрабатывается в своем потоке.

timer3.Stop();

Console.WriteLine($"Время выполнения Parallel.Foreach: {timer3.Elapsed.TotalMilliseconds}");

Stopwatch timer4 = new Stopwatch();

timer4.Start();

foreach (int i in arr2)

{

createMas3(i);////то цикл foreach, который перебирает все элементы массива arr2.

// Для каждого элемента массива вызывается функция createMas3(i), которая изменяет значение переменной i, добавляя 5

}

timer4.Stop();

Console.WriteLine($"Время выполнения foreach: {timer4.Elapsed.TotalMilliseconds}");

/////////////////////////////////////////////////////////////////

void method()

{

decimal sum = 0;

for (int i = 0; i < 1000000; i++)

{

sum += i; //Это метод, который выполняет суммирование чисел от 0 до 999,999

}

}

void method2()

{

int mul = 1;

for (int i = 1; i <= 50; i++)//Это метод, который вычисляет произведение чисел от 1 до 50.

{

mul \*= i;

}

}

Stopwatch timer5 = new Stopwatch(); // отслеживать время выполнения

timer5.Start();

//() => method() — анонимная функция, которая вызывает метод method().

Parallel.Invoke(() => method(), () => method2()); //Parallel.Invoke — это метод из библиотеки System.Threading.Tasks, который позволяет выполнять несколько действий (методов, делегатов или лямбда-выражений) параллельно, то есть одновременно в разных потоках.

timer5.Stop();

Console.WriteLine($"Время работы Parallel Invoke {timer5.Elapsed.TotalMilliseconds}");

//////////////////////////////////////////////////////////////////////

async static void quest\_8() //Ключевое слово async обозначает, что метод будет асинхронным.

{

using var writer = new StreamWriter("C:\\Users\\User\\Documents\\2курс1сем\\лабыООП\\lab15\\output.txt"); //Создается объект StreamWriter, который используется для записи данных в файл.

Console.WriteLine("Начало асинхронной записи в файл...");

//await — это ключевое слово, которое приостанавливает выполнение метода, в котором оно используется, до тех пор, пока не завершится асинхронная операция

//await не блокирует поток целиком. Он приостанавливает только выполнение кода внутри метода quest\_8().

await writer.WriteLineAsync("Какая-то асинхронаая запись, которая не мешает выполнению основной программы!"); //Метод WriteLineAsync записывает строку в файл асинхронно (то есть без блокировки основного потока программы).

Console.WriteLine("Запись в файл завершена");

}

quest\_8();

////////////////////////////////

BlockingCollection<string> sklad = new BlockingCollection<string>(5); //BlockingCollection — это потокобезопасная коллекция.блокируя операции добавления и извлечения в зависимости от состояния коллекции (например, когда она заполнена или пуста).

void postav(string prod, int time) //работа поставщика)

{

while (true)

{

sklad.Add(prod); //// Добавляем товар на склад

Console.WriteLine($"Поставщик доставил {prod}");

skladState();//// Показываем текущее состояние склада

Thread.Sleep(time); // // Поставщик ожидает перед тем, как добавить новый товар

}

}

void Client(int clientID) //метод моделирует поведение клиента, который пытается купить товары с некого склада. Он выполняется бесконечно (так как находится в цикле while(true)), пока клиент не остановится

{

while (true)

{

Random random = new Random(); //создается объект random, который будет использоваться для генерации случайных чисел. Это нужно, чтобы клиент случайным образом выбирал время ожидания перед тем, как попытаться купить товар.

Thread.Sleep(random.Next(0, 5000)); //В этой строке клиент "ждет" случайное время

if (sklad.TryTake(out string prod)) //Пытается удалить элемент из BlockingCollection<T>.Этот метод пытается извлечь товар из коллекции sklad.

{

Console.WriteLine($"Клиент с ID = {clientID} купил {prod}"); //Если товар есть на складе, то операция проходит успешно, и товар записывается в переменную prod.

skladState();

}

else

{

Console.WriteLine($"Клиент ушел с пустыми руками");

}

}

}

void skladState()

{

Console.WriteLine($"Текущее состояние склада: {string.Join(", ", sklad)}"); // создает строку, в которой элементы коллекции sklad будут разделены запятой. Таким образом, мы получаем строку с перечнем всех товаров на складе.

}

var postavschiki = new List<Task>() { //Здесь создается список postavschiki, который будет содержать задачи, выполняющиеся параллельно.

Task.Run(() => postav("Телевизор", 2500)), //Каждый элемент списка — это задача, запускаемая с помощью Task.Run(), которая вызывает метод postav с разными товарами и разным временем задержки

Task.Run(() => postav("Компьютер", 3500)),

Task.Run(() => postav("Планшет", 5000)),

Task.Run(() => postav("Телефон", 1000)),

Task.Run(() => postav("Ноутбук", 4500))

};

List<Task> clients = new List<Task>(); //Здесь создается список clients, который будет хранить объекты типа Task. Task представляет собой задачу, которая выполняется асинхронно (параллельно) в одном или нескольких потоках.

for (int i = 0; i < 10; i++) //

{

clients.Add(Task.Run(() => Client(i)));//Этот цикл создает 10 параллельных задач (по одной для каждого клиента), которые будут выполняться одновременно (параллельно) с другими задачами. Каждая задача (клиент) будет выполнять метод Client, в котором будут описаны действия клиента с товаром (например, покупка товара из склада).

}

foreach (Task task in clients)//для каждой задачи из списка программа ожидает, пока она завершится

{

task.Wait(); // Использование task.Wait() заставляет основной поток дождаться завершения всех задач, то есть гарантирует, что программа не завершится, пока все клиенты не закончат свою работу.

}

//////////////////////////////////////////////////

}

}

}