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

Библиотека параллельных задач TPL (Task Parallel Library), основной функционал которой располагается в пространстве имен System. Threading. Tasks. Данная библиотека позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах, если на целевом компьютере имеется несколько ядер.

Сlass Task упрощает написание кода для создания новых потоков.Поэтому начиная с .NET 4.0. рекомендуется использовать именно класс Task для создания многопоточных приложений.

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

Без распараллеливания не обойтись при программировании алгоритмов решения тех задач, которые в принципе не могут быть решены на однопроцессорных системах. Это может проявиться в двух случаях: либо когда для решения задачи требуется слишком много времени, либо когда для программы недостаточно оперативной памяти на однопроцессорной системе. Для небольших задач зачастую оказывается, что параллельная версия работает медленнее, чем однопроцессорная.

1. **В чем основные достоинства работы с задачами по сравнению с потоками?**

Простота доступности потоков к памяти процесса имеет свои недостатки.

• Потоки могут легко разрушить адресное пространство процесса.

• Потоки необходимо синхронизировать при параллельном доступе (для чтения или записи) к памяти.

• Один поток может ликвидировать целый процесс или программу.

• Потоки существуют только в рамках единого процесса и, следовательно, не являются многократно используемыми.

Идея основанного на задачах параллизма состоит в том, чтобы разбить работу на небольшие задачи, которые могут дать результаты в будущем, а затем эффективно распределить ровно столько потоков, сколько есть процессоров, чтобы вы не тратили время на переключение контекста.

1. **Приведите три способа создания и/или запуска Task?**

1. .Start()

2. .Run()

3. TaskFactory.StartNew(Action<Object>, Object)

4. .RunSynchronously()

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

Task.Wait() - Ожидает завершения выполнения задачи Task.

Task.WaitAll(Task[]) - Ожидает завершения выполнения всех указанных объектов Task.

Task.WaitAny(Task[]) - Ожидает завершения выполнения любого из указанных объектов Task.

1. **Приведите пример синхронного запуска Task?**

Task t = new Task(()=> {Console.WriteLine(“SAS”);});

t.RunSynchronously();

1. **Как создать задачу с возвратом результата?**

int sum(int a,int b,int c){ return a + b + c;}

Task<int> result = new Task<int>(() => sum(rand\_num1.Result, rand\_num2.Result, rand\_num3.Result));

result.Start();

WriteLine($"Результат вычислений задачи {result.Id}:{result.Result} ");

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

Для обработки ошибок выражение **await** помещается в блок try.

При возникновении ошибки у объекта **Task**, представляющего асинхронную задачу, в которой произошла ошибка, свойство **IsFaulted** имеет значение true. Кроме того, свойство **Exception** объекта **Task** содержит всю информацию об ошибке.

Можно воспользоваться свойством **task.Exception.InnerException.Message**.

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

Для отмены асинхронных операций используются классы **CancellationToken** и **CancellationTokenSource**.

**CancellationToken** содержит информацию о том, надо ли отменять асинхронную задачу. Асинхронная задача, в которую передается объект **CancellationToken**, периодически проверяет состояние этого объекта. Если его свойство **IsCancellationRequested** равно **true**, то задача должна остановить все свои операции.

Для создания объекта **CancellationToken** применяется объект **CancellationTokenSource**. Кроме того, при вызове у **CancellationTokenSource** метода **Cancel**() у объекта **CancellationToken** свойство **IsCancellationRequested** будет установлено в **true**.

1. **Как организовать задачу продолжения (continuation task)?**

Задачи продолжения или **continuation** **task** позволяют определить задачи, которые выполняются после завершения других задач. Задача продолжения задается с помощью метода **ContinueWith**, который в качестве параметра принимает делегат **Action<Task>.**

Task<int> task1 = new Task<int>(() => Sum(4, 5));

Task task2 = task1.ContinueWith(sum => Display(sum.Result));

task1.Start();

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

Объект задачи, также имеет определенные условия, чтобы к нему можно было применить await:

* Ожидаемый тип должен иметь публичный (или **internal**) метод **GetAwaiter**(), это может быть и метод расширения. Этот метод возвращает *объект ожидания*
* Объект ожидания должен реализовать интерфейс **INotifyCompletion**, который обязывает реализовать метод **void OnCompleted(Action continuation**). Также он должен иметь экземплярные свойство **bool** **IsCompleted**, метод **void** **GetResult**(). Может быть как структурой, так и классом.

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

Класс **Parallel** также является частью TPL и предназначен для упрощения параллельного выполнения кода. **Parallel** имеет ряд методов, которые позволяют распараллелить задачу.

Одним из методов, позволяющих параллельное выполнение задач, является метод **Invoke.**

Метод **Parallel.For** позволяет выполнять итерации цикла параллельно.

Метод **Parallel.ForEach** осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable, подобно циклу foreach, только осуществляет параллельное выполнение перебора.

Методы **Parallel.ForEach** и **Parallel.For** возвращают объект **ParallelLoopResult**, наиболее значимыми свойствами которого являются два следующих:

* **IsCompleted**: определяет, завершилось ли полное выполнение параллельного цикла
* **LowestBreakIteration**: возвращает индекс, на котором произошло прерывание работы цикла

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

Parallel.For(0, arr\_1.Length, (int i) =>

{

arr\_1[i] = new int[1000000];

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

{

arr\_1[i][item] = rand.Next(0, 100);

}

});

1. **Приведите пример задачи с Parallel.ForEach**

decimal factorial(int x)

{

decimal result = 1;

for (decimal i = 1; i <= x; i++)

{

result \*= i;

}

return result;} Parallel.ForEach(list, (a) => .WriteLine("Факториал числа " + a + " равен " + factorial(a)));

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

Parallel.Invoke

(

() => { Thread.Sleep(1000); WriteLine($"Выполняется задача {Task.CurrentId}"); Thread.Sleep(3000); },

() => { Thread.Sleep(1000); WriteLine($"Выполняется задача {Task.CurrentId}"); Thread.Sleep(3000); },

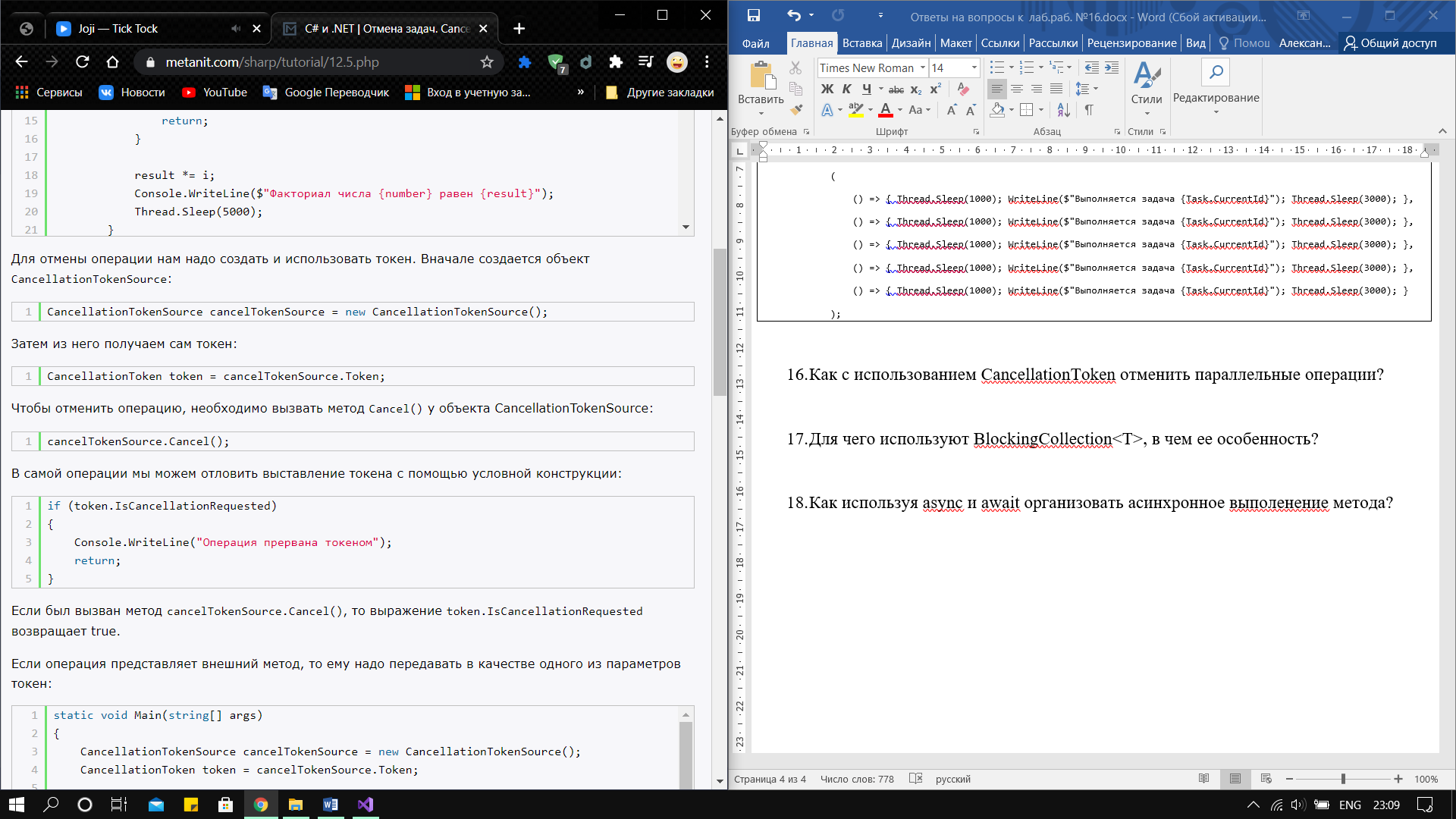
() => { Thread.Sleep(1000); WriteLine($"Выполняется задача {Task.CurrentId}"); Thread.Sleep(3000); },

() => { Thread.Sleep(1000); WriteLine($"Выполняется задача {Task.CurrentId}"); Thread.Sleep(3000); },

() => { Thread.Sleep(1000); WriteLine($"Выполняется задача {Task.CurrentId}"); Thread.Sleep(3000); }

);

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



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

Коллекция, которая осуществляет блокировку и ожидает, пока не появится возможность выполнить действие по добавлению или извлечению элемента. **BlockingCollection**<T> предлагает интерфейс для добавления и извлечения элементов методами **Add**() и **Take**(). Эти методы блокируют поток и затем ожидают, пока не появится возможность выполнить задачу.

Метод **Add**() имеет перегрузку, которой можно также передать **CancellationToken**. Эта лексема всегда отменяет блокирующий вызов.

Если не нужно, чтобы поток ожидал бесконечное время, и не хотите отменять вызов извне, доступны также методы **TryAdd**() и **TryTake**(). В них можно указать значение таймаута — максимального периода времени, в течение которого вы готовы блокировать поток и ждать, пока вызов не даст сбой.

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

**Асинхонный метод** обладает следующими признаками:

* В заголовке метода используется модификатор **async**
* Метод содержит одно или несколько выражений **await**
* В качестве возвращаемого типа используется один из следующих:
  + void
  + Task
  + Task<T>
  + ValueTask<T>

Асинхронный метод, как и обычный, может использовать любое количество параметров или не использовать их вообще. Однако асинхронный метод не может определять параметры с модификаторами **out** и **ref**.

Также стоит отметить, что слово **async**, которое указывается в определении метода, не делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений **await**.