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

**Task Parallel Library** – позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах (для создания многопоточных приложений)

**класс Task** – опис. отд. задачу, кот. запуск. асинхронно в одном из потоков из пула потоков  
(м. запуск. синхронно в тек. потоке)

везде в кач. параметра делегат Action

2. Почему эффект от распараллеливания наблюдается на большом кол-ве элементов?

3. В чем осн. достоинства работы с задачами по сравнению с потоками?

в Thread:  
1) нет механизма продолжений  
2) затруднено получение значения результата из потока  
3) повыш. расход памяти и замедление работы приложения

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

1) Task task = new Task(() = > Console.WriteLine(“Hello!”)); //3  
task.Start()

2) Task task = Task.Factory.StartNew(() => Console.WriteLine(“Hello!”); //1  
3) Task task = Task.Run(() => Console.WriteLine(“Hello!”); //2

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

**Wait()** – приостан. тек. поток до завершения задачи  
**WaitAll()** – стат., приост. тек. поток до завершения всех указ. задач  
**WaitAny()** – стат., приост. тек поток до завершения любой из указ. задач

6. Приведите пример синхронного запуска Task?

Action <object> method = x => Console.WriteLine(“yo”);  
var task4 = new Task( method, TaskCreationOptions.LongRunnig);  
task4. RunSynchronously();

7. Как создать задачу с возвратом результата?

Task<TResult> - опис. задачу, возвр. значение типа TResult

приним. аргументы типа:  
Func<TResult>  
Func<object, TResult>

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

Task task5 = Task.Run(() => { throw new Exception() });  
try {  
 task5.Wait();  
}  
catch (AggregateException ex) {  
 var message = ex.InnerException.Message;  
 Console.WriteLine(message);  
}

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

CancellationTokenSource tokenSource = new CancellationTokenSource();  
//исп. токен в двух задачах  
new Task(method, tokenSource.Token).Start();  
new Task(method, tokenSource.Token).Start();

//отменяем задачи  
tokenSource.Cancel()

10. Как организовать задачу продолжения (continuation task)?

сообщ. задаче, что после ее завершения она д. продолжить делать что-то другое

1) Task task6 = Task.Run(() => Console.Write(“Doing..”));  
Task task7 = task6.ContinueWith (t => Console.Write(“Continuation”);

2) Task task8 = Task.Run(() => Console.Write(“One…”);  
Task task9 = Task.Run(() => Console.Write(“Two…”);  
Task continuation = Task.WhenAll(task8, task9).  
 ContinueWith ( t => Console.WriteLine(“Three..”);

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

Объект ожидания – любой объект, имеющий методы

адача продолжения (также называемая просто продолжением) — это асинхронная задача, вызываемая другой задачей, которая называется предшествующей, при завершении этой предшествующей задачи.

Хотя продолжения относительно легко использовать, они представляют собой мощный и гибкий инструмент. Например, с их помощью можно выполнять следующее.

Передавать данные из предшествующей задачи в продолжение.

Указывать точные условия, при которых продолжение будет вызываться или не будет вызываться.

Отменять продолжение перед его запуском либо совместно с его выполнением.

Определять подсказки, как должно планироваться продолжение.

Вызывать несколько продолжений из одной и той же предшествующей задачи.

Вызывать одно продолжение по завершении всех или одной из нескольких предшествующих задач.

Прикреплять продолжения одно после другого до любой произвольной длины.

Использовать продолжение для обработки исключений, вызванных предшествующей задачей.  
OnCompleted() + GetResult() + св-во IsCompleted

Task<int> what = Task.Run(() => Enumerable. Range(1, 100000)   
 .Count(n => (n%2==0)));  
//получаем объект продолжения  
var awaiter = what.GetAwaiter();

//что делать после оконч. предшественника  
awaiter.OnCompleted(() => {  
//получаем рез. вычислений предшественника  
 int res = awaiter.GetResult();  
 Console.WriteLine(res);

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

позв. распараллеливать циклы и посл-ст блоков кода  
For(), ForEach() //паралл. аналоги циклов for, foreach  
..Invoke() – шаблоны (на задачах, поддерж. искл. и токен отмены)

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

Parallel.For (1, 10, z => //1,10 – нач,кон знач счетчика,  
 { int r=1; //z – тело цикла в виде объекта делегата  
 for (int y=1; y<=10; y++) // 3 4 6 8 9 2 5 1 7  
 r \*= z;  
 }

Paralle.For (1, 10, (int z, ParallelLoopState pd) => {  
 Console.WriteLine(z);  
 int r=1;  
 for (int y=1; y<=10; y++) // 3 4 6 8 9 1 2 7 5  
 r\*=z;

Поддерж. императивность – оп-р, след. за вызовом метода вызван после заверш. всех задач

14. Приведите пример задачи с Parallel.ForEach

ParallelLoopResult listFact = Parallel.ForEach<int>  
 (new List<int>() { 1, 3, 5, 8 }, //коллекция  
 Factorial); //делегат, вып. раз за итерацию  
. //перебир. эл-та коллекции

15. Приведите пример с Parallel.Invoke()

позв. распараллелить исп. блоков операторов – набор задач, кот. вып. в одном потоке  
их могут запуск одновременно

Parallel.Invoke(() => new WebClient().DownloadFile(“http…”),  
 () => new WebClient().DownloadFile(“http…”));

16. Как с использованием CancellationToken отменить парал. операции?

CancellationTokenSource ctks = new CancellationTokenSource();  
CancellationToken token = ctks.Token;

Parallel.ForEach<int>(  
 new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8},  
 new ParallelOptions { CancellationToken = token},  
 Factorial);

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

используется в качестве оболочки для IProducerConsumerCollection<**T**> экземпляра и позволяет выполнять попытки удаления из коллекции, пока данные не будут доступны для удаления.

18. Как используя async и await организовать асинхронное вып. метода?

При асинхр. вызове поток вып-ния раздел. на 2 части:  
в одной – вып. метод  
в другой – процесс программы

async – указ., что д-й метод м. содерж. 1/неск выражений await

//асинхр. метод  
static async void FactorialAsync() {  
 Console.Write(“Начало”); //вып. синхронно  
 await Task.Run(() => Factorial\_Six()); //вып. асинхронно  
 Console.Write(“Конец”) //вып синхронно

main()  
 FactorialAsync();  
 Console.Write(“Квадрат 3” + 9);

Асинхр. задача, кот. может выполнятся долго, не блокирует метод Main,  
и мы можем продолжать работу с ним  
когда асинхр. задача заверш. (await) Fact\_Six,  
продолж. асинхр. метод FactAsync  
ост. часть этого метода после await вып. Синхронно

**Асинхонный метод** обладает следующими признаками:

* В заголовке метода используется модификатор **async**
* Метод содержит одно или несколько выражений **await**

ыражение **await** определяет задачу, которая будет выполняться асинхронно.