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

**TPL (Task Parallel Library)** — это библиотека .NET для простого и эффективного выполнения параллельных операций. Основной элемент TPL — это класс Task, представляющий асинхронную задачу.

**Использование Task:**

* Выполнение фоновых операций.
* Упрощение работы с асинхронностью.
* Планирование и управление задачами.

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

* При небольшом объеме данных накладные расходы на управление потоками, создание задач и синхронизацию превышают выигрыш от параллельного выполнения.
* На большом количестве элементов удается полностью загрузить процессоры, что уменьшает относительные затраты на управление и увеличивает производительность.

**3. Основные достоинства работы с задачами по сравнению с потоками:**

* **Упрощение кода:** Task поддерживает асинхронный стиль программирования (async/await).
* **Легковесность:** Пул потоков TPL автоматически перераспределяет задачи, избегая затрат на создание потоков.
* **Продвинутые возможности:** Задачи поддерживают продолжения (ContinueWith), отмену (CancellationToken) и агрегированные операции (Task.WhenAll).

**4. Три способа создания и/или запуска Task:**

1. **С использованием конструктора Task и метода Start:**
2. Task task = new Task(() => Console.WriteLine("Задача выполнена"));
3. task.Start();
4. **С использованием Task.Run:**
5. Task.Run(() => Console.WriteLine("Задача выполнена"));
6. **С использованием Task.Factory.StartNew:**
7. Task.Factory.StartNew(() => Console.WriteLine("Задача выполнена"));

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

* **Wait:** Блокирует выполнение текущего потока до завершения задачи.
* **WaitAll:** Ожидает завершения всех переданных задач.
* **WaitAny:** Ожидает завершения любой из переданных задач.

**6. Пример синхронного запуска Task:**

Task task = Task.Run(() => Console.WriteLine("Задача выполнена"));

task.Wait(); // Синхронное ожидание завершения

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

Используется Task<T>, где T — тип возвращаемого значения.

Task<int> task = Task.Run(() => 42);

int result = task.Result; // Получение результата

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

Исключения обрабатываются с помощью блока try-catch и свойства AggregateException.

try

{

Task task = Task.Run(() => throw new InvalidOperationException("Ошибка"));

task.Wait();

}

catch (AggregateException ex)

{

foreach (var inner in ex.InnerExceptions)

Console.WriteLine(inner.Message);

}

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

* **CancellationToken** — структура для уведомления о необходимости отмены задачи.
* Используется с CancellationTokenSource.

Пример:

var cts = new CancellationTokenSource();

var token = cts.Token;

Task.Run(() =>

{

while (!token.IsCancellationRequested)

{

Console.WriteLine("Работаю...");

Thread.Sleep(500);

}

}, token);

Thread.Sleep(2000);

cts.Cancel(); // Отмена задачи

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

Используется метод ContinueWith.

Task task = Task.Run(() => Console.WriteLine("Первая задача"));

task.ContinueWith(t => Console.WriteLine("Задача продолжения"));

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

Объект ожидания (TaskAwaiter) используется для обработки результата задачи асинхронно.

Task<int> task = Task.Run(() => 42);

var awaiter = task.GetAwaiter();

awaiter.OnCompleted(() =>

{

int result = awaiter.GetResult();

Console.WriteLine($"Результат: {result}");

});

**12. Назначение класса System.Threading.Tasks.Parallel:**

Класс Parallel предоставляет высокоуровневый API для параллельного выполнения циклов и блоков кода, упрощая работу с потоками.

**13. Пример задачи с Parallel.For:**

Parallel.For(0, 10, i =>

{

Console.WriteLine($"Итерация {i} выполняется");

});

**14. Пример задачи с Parallel.ForEach:**

var items = Enumerable.Range(1, 10);

Parallel.ForEach(items, item =>

{

Console.WriteLine($"Элемент {item} обрабатывается");

});

**15. Пример с Parallel.Invoke:**

Parallel.Invoke(

() => Console.WriteLine("Задача 1"),

() => Console.WriteLine("Задача 2"),

() => Console.WriteLine("Задача 3")

);

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

var cts = new CancellationTokenSource();

var token = cts.Token;

Task.Run(() =>

{

Parallel.For(0, 100, new ParallelOptions { CancellationToken = token }, i =>

{

Console.WriteLine($"Итерация {i}");

Thread.Sleep(100);

});

});

Thread.Sleep(1000);

cts.Cancel();

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

**BlockingCollection<T>**:

* Используется для потокобезопасного обмена данными между потоками.
* Поддерживает операции блокировки при добавлении или удалении элементов.

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

static async Task<int> CalculateAsync()

{

await Task.Delay(1000); // Имитируем длительную операцию

return 42;

}

static async Task Main(string[] args)

{

int result = await CalculateAsync();

Console.WriteLine($"Результат: {result}");

}