Лекция 8
Многопоточность vol. 2

В прошлой части

- Понятия процесс и поток
- Работа с потоками в C#. Класс Thread
- Синхронизация потоков
- Ситуации Deadlock
- GP GPU

Task Parallel Library

- В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов .NET задача представлена специальным классом - классом **Task**, который находится в пространстве имен **System.Threading.Tasks**. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

```
Task task = new Task(() => Console.WriteLine("Hello Task!"));
task.Start();
Task task = Task.Run(() => Console.WriteLine("Hello Task!"));
Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));
```

TaskSheduler

- Планировщик задач представляет объект, обрабатывающий низкоуровневую постановку задач в очередь на потоки. Планировщик по умолчанию для библиотеки параллельных задач использует пул потоков платформы .NET Framework.

```
var lcts = new LimitedConcurrencyLevelTaskScheduler(2);
var factory = new TaskFactory(lcts);
var t = factory.StartNew(() => Thread.Sleep(3000))
```

Свойства класса Task

- ✔ AsyncState: возвращает объект состояния задачи
- ✔ CurrentId: возвращает идентификатор текущей задачи
- ✔ Exception: возвращает объект исключения, возникшего при выполнении задачи
- ✔ Status: возвращает статус задачи
  - Created - 0 - Задача инициализирована, но еще не запланирована
  - WaitingForActivation - 1 - Задача ожидает активации и внутреннего планирования инфраструктурой .NET
  - WaitingToRun - 2 - Задача запланирована для выполнения, но еще не начала выполняться
  - Running - 3 - Задача выполняется, но еще не завершилась
  - WaitingForChildrenToComplete - 4 - Задача закончила выполнение и неявно ожидает завершения подключенных к ней дочерних задач
  - RanToCompletion - 5 - Задача успешно завершена
  - Canceled - 6 - Задача приняла отмену, создав исключение OperationCanceledException с собственным токеном CancellationToken
  - Faulted - 7 - Задача завершилась из-за необработанного исключения

```
class CustomData{
        public long CreationTime;
        public int Name;
        public int ThreadNum;
}
…
Task[] taskArray = new Task[10];
for (int i = 0; i < taskArray.Length; i++) {
        taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                CustomData data = obj as CustomData;
                if (data == null) return;
                data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
        },
        new CustomData() {Name = i, CreationTime = DateTime.Now.Ticks} );
}
Task.WaitAll(taskArray);
foreach (var task in taskArray) {
        var data = task.AsyncState as CustomData;
        if (data != null)
        Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                data.Name, data.CreationTime, data.ThreadNum);
        }
}
```

TaskCreationOptions

- AttachedToParent – 4 -Указывает, что задача присоединена к родительской задаче в иерархии задач
- DenyChildAttach – 8 - Указывает, что любая дочерняя задача, для которой выполняется попытка выполнения в качестве подсоединенной дочерней задачи (т. е. она создается с параметром AttachedToParent), не сможет подключиться к родительской задаче и будет выполняться как отсоединенная дочерняя задача
- HideScheduler -16 - Не позволяет видеть внешний планировщик как текущий планировщик в созданной задаче. Это означает, что такие операции, как StartNew или ContinueWith, которые выполняются в созданной задаче, в качестве текущего планировщика будут видеть свойство Default.
- LongRunning – 2 - Указывает, что задача будет выполняться долго в качестве общей операции, включающей еще несколько компонентов, по размеру превышающих детализированные системы. Предоставляет сведения для TaskScheduler, что следует ожидать избыточной подписки. Он также подсказывает планировщику задач, что для задачи может потребоваться дополнительный поток, чтобы она не блокировала дальнейший ход работы других потоков или рабочих элементов в локальной очереди пула потоков
- None – 0 - Указывает, что следует использовать поведение по умолчанию.
- PreferFairness – 1 - Рекомендация для TaskScheduler для планирования задач максимально прямым способом, то есть задачи, запланированные ранее, будут выполняться ранее, а более поздние — позднее.
- RunContinuationsAsynchronously - 64 - Принудительное асинхронное выполнение продолжений, добавляемых в текущую задачу

Возвращение результатов из Task

```
Task<int> task1 = new Task<int>(() => Factorial(5));
task1.Start();
Console.WriteLine($"Факториал числа 5 равен {task1.Result}");

Var task2 = new Task<Book>(() => {
    return new Book { Title = "Война и мир", Author = "Л. Толстой" };
});
task2.Start();
var b = task2.Result;
Console.WriteLine($"Название книги: {b.Title}, автор: {b.Author}");
```

Задачи продолжения

```
var task1 = new Task(() => {
    Console.WriteLine("Id задачи: {0}", Task.CurrentId);
});
var task2 = task1.ContinueWith((Task t) =>{
    Console.WriteLine("Id задачи: {0}", Task.CurrentId);
});
var task3 = task2.ContinueWith((Task t) =>{
    Console.WriteLine("Id задачи: {0}", Task.CurrentId);
});
var task4 = task3.ContinueWith((Task t) =>{
    Console.WriteLine("Id задачи: {0}", Task.CurrentId);
});
task1.Start();
```

Управление задачами

- Task.WhenAll
- Task.WhenAny
- Task.Delay
- Task(T).FromResult - создает Task<TResult>, которая завершается удачно с указанным результатом (например, при кэшировании данных в приложении)

Класс 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, вызывающий параллельно переданные методы

```
Parallel.Invoke(Display,
    () => {
        Console.WriteLine("Выполняется задача {0}", Task.CurrentId);
        Thread.Sleep(3000);
    },
    () => Factorial(5));
```

Parallel.For

- Метод Parallel.For позволяет выполнять итерации цикла параллельно. Он имеет следующее определение: For(int, int, Action<int>), где первый параметр задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию. На выходе метод возвращает структуру ParallelLoopResult, которая содержит информацию о выполнении цикла:
- IsCompleted - получает значение, указывающее, дошел ли цикл до завершения, то есть все итерации цикла выполнены и он не получил запроса на преждевременное прерывание работы
- LowestBreakIteration - получает индекс нижней итерации, из которой был вызван метод Break()

```
Parallel.For(1, 10, Factorial);
```

Parallel.ForEach

- Метод Parallel.ForEach осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable, подобно циклу foreach, только осуществляет параллельное выполнение перебора. Он имеет следующее определение: ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> source, Action<TSource> body), где первый параметр представляет перебираемую коллекцию, а второй параметр - делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции. На выходе метод возвращает структуру ParallelLoopResult

```
var result = Parallel.ForEach<int>(new List<int>() { 1, 3, 5, 8 },
Factorial);
```

Отмена задач

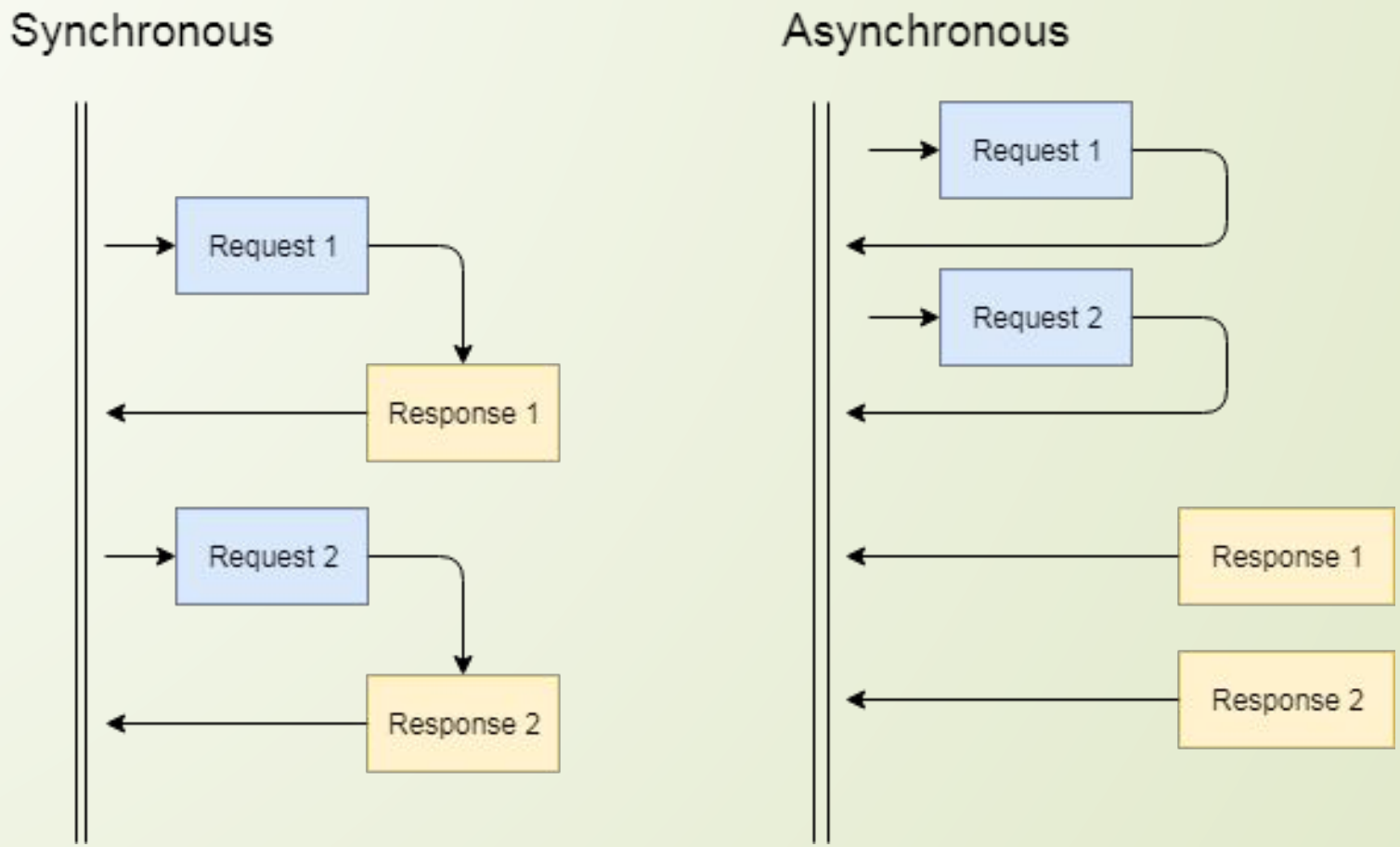
```
var tokenSource = new CancellationTokenSource();
var ct = tokenSource.Token;
var task = Task.Run(() => {
      bool moreToDo = true;
      while (moreToDo){
            …
            if (ct.IsCancellationRequested){
                  ct.ThrowIfCancellationRequested();
            }
      }
}, ct);
…
tokenSource.Cancel();
```

Параллельные коллекции

- Классы коллекций в пространства имен System.Collections.Concurrent поддерживают потокобезопасные операции добавления и удаления, которые избегают блокировок везде, где это возможно, и применяют только детально настроенные блокировки. Класс параллельных коллекций не требует использовать блокировки в пользовательском коде для доступа к элементам. Классы параллельных коллекций могут значительно повысить производительность по сравнению с типами System.Collections.ArrayList и System.Collections.Generic.List<T> (где блокировка реализуется пользователем) в сценариях одновременного добавления и удаления элементов коллекции из нескольких потоков.

- IProducerConsumerCollection<T> - это интерфейс, тип которого должен быть реализован для использования в классе BlockingCollection
- BlockingCollection<T> Предоставляет возможности блокировки и ограничения для всех типов, реализующих интерфейс IProducerConsumerCollection<T>
- ConcurrentDictionary<TKey,TValue> - потокобезопасная реализация словаря пар "ключ-значение".
- ConcurrentQueue<T> - потокобезопасная реализация очереди с типом "первым поступил — первым обслужен" (FIFO)
- ConcurrentStack<T> - потокобезопасная реализация стека с типом "последним поступил — первым обслужен" (LIFO)
- ConcurrentBag<T> - потокобезопасная реализация неупорядоченной коллекции элементов

Асинхронное программирование



Асинхронное программирование на основе делегатов

```
DisplayHandler handler = new DisplayHandler(Display);
IAsyncResult resultObj = handler.BeginInvoke(10, new AsyncCallback(AsyncCompleted), "Асинхронные вызовы");
Console.WriteLine("Продолжается работа метода Main");
int res = handler.EndInvoke(resultObj);
Console.WriteLine("Результат: {0}", res);
...
static int Display(int k){
        Console.WriteLine("Начинается работа метода Display....");
        int result = 0;
        for (int i = 1; i < 10; i++) result += k * i;
        Thread.Sleep(3000);
        Console.WriteLine("Завершается работа метода Display....");
        return result;
}
static void AsyncCompleted(IAsyncResult resObj) {
        string mes = (string)resObj.AsyncState;
        Console.WriteLine(mes);
        Console.WriteLine("Работа асинхронного делегата завершена");
}
```

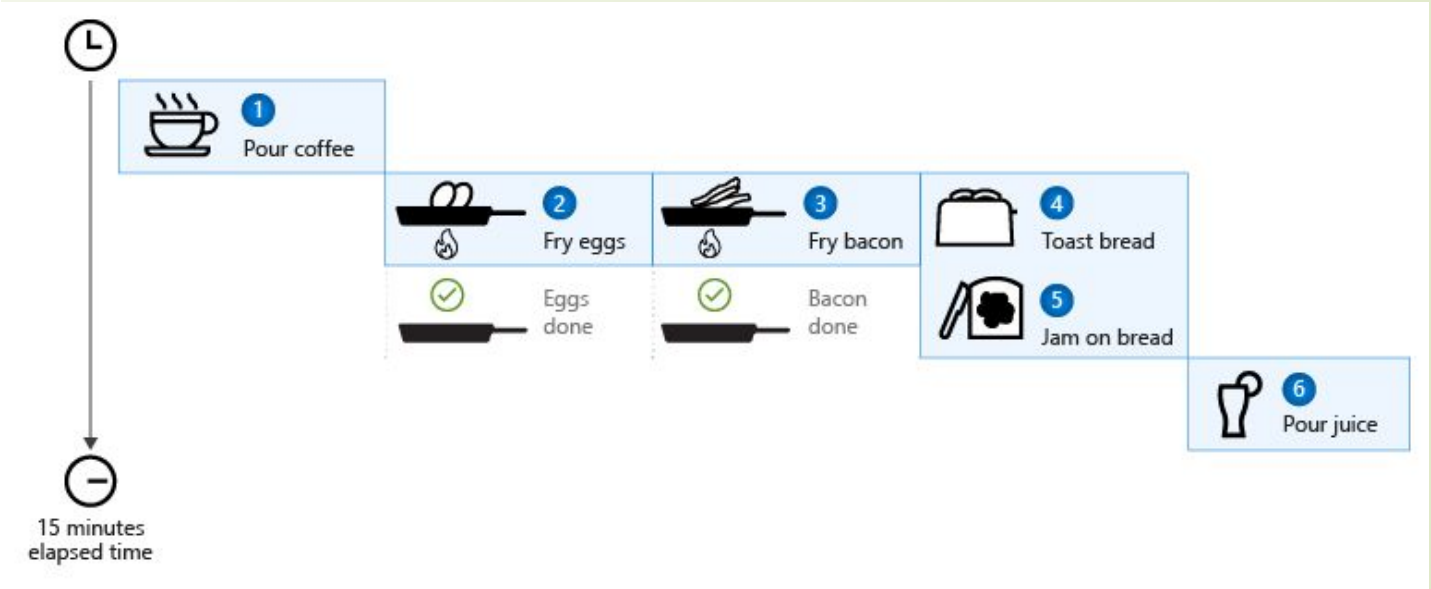
- В .NET 4.5 во фреймворк были добавлены два новых ключевых слова async и await, цель которых - упростить написание асинхронного кода. Вместе с функциональностью задач Task они составляют основу новой модели асинхронного программирования в .NET, которая называется Task-based Asynchronous Pattern. Операторы async и await используются вместе для создания асинхронного метода. Такой метод, определенный с помощью модификатора async и содержащий одно или несколько выражений await, называется асинхронным методом.
- Ключевое слово async указывает, что метод или лямбда-выражение являются асинхронными. А оператор await применяется к задаче в асинхронных методах, чтобы приостановить выполнение метода до тех пор, пока эта задача не завершится. При этом выполнение потока, в котором был вызван асинхронный метод, не прерывается.

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number) {
        var toast = await ToastBreadAsync(number);
        ApplyButter(toast);
        ApplyJam(toast);
        return toast;
}
private static Juice PourOJ() {
        Console.WriteLine("Pouring orange juice");
        return new Juice();
}
private static void ApplyJam(Toast toast) =>
        Console.WriteLine("Putting jam on the toast");
private static void ApplyButter(Toast toast) =>
        Console.WriteLine("Putting butter on the toast");
private static async Task<Toast> ToastBreadAsync(int slices){
        for (int slice = 0; slice < slices; slice++) {
            Console.WriteLine("Putting a slice of bread in the toaster");
        }
        Console.WriteLine("Start toasting...");
        await Task.Delay(3000);
        return new Toast();
}
```

```
private static async Task<Bacon> FryBaconAsync(int slices)
    {
        Console.WriteLine($"putting {slices} slices of bacon in the pan");
        Console.WriteLine("cooking first side of bacon...");
        await Task.Delay(3000);
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("flipping a slice of bacon");
        }
        Console.WriteLine("cooking the second side of bacon...");
        await Task.Delay(3000);
        Console.WriteLine("Put bacon on plate");

        return new Bacon();
    }

    private static async Task<Egg> FryEggsAsync(int howMany)
    {
        Console.WriteLine("Warming the egg pan...");
        await Task.Delay(3000);
        Console.WriteLine($"cracking {howMany} eggs");
        Console.WriteLine("cooking the eggs ...");
        await Task.Delay(3000);
        Console.WriteLine("Put eggs on plate");

        return new Egg();
    }

    private static Coffee PourCoffee()
    {
        Console.WriteLine("Pouring coffee");
        return new Coffee();
    }
```

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
var eggsTask = FryEggsAsync(2);
var baconTask = FryBaconAsync(3);
var toastTask = MakeToastWithButterAndJamAsync(2);
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0){
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask) {
        Console.WriteLine("eggs are ready");
    }
    else if (finishedTask == baconTask){
        Console.WriteLine("bacon is ready");
    }
    else if (finishedTask == toastTask){
        Console.WriteLine("toast is ready");
    }
    breakfastTasks.Remove(finishedTask);
}
Juice oj = PourOJ();
Console.WriteLine("oj is ready");
Console.WriteLine("Breakfast is ready!");
```


1 Pour coffee
2 Heat pan
3 Fry eggs
4 Fry bacon
5 Toast bread
6 Jam on bread
7 Pour juice
30 minutes elapsed time


1 Pour coffee
2 Fry eggs
Eggs done
3 Fry bacon
Bacon done
4 Toast bread
5 Jam on bread
6 Pour juice
15 minutes elapsed time

- Постоянное использование блокирующих вызовов нивелирует все преимущества асинхронного программирования превращая его в синхронное