МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания

по выполнению лабораторных работ
по дисциплине «ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ»
для студентов направления подготовки
09.03.03 Прикладная информатика

(ЭЛЕКТРОННЫЙ ДОКУМЕНТ)

Ставрополь 2026

Оглавление

Варианты заданий

В течение семестра вам необходимо разработать приложение, соответствующее вашей лабораторной работе. В каждой лабораторной работе вам необходимо дополнять приложение, добавляя в него функционал, соответствующий тематике лабораторной работы.

1. Разработка музыкального плеера.
2. Разработка приложения для просмотра изображений
3. Игра односторонний морской бой
4. Инженерный калькулятор
5. Калькулятор программиста
6. Банковский кредитный калькулятор
7. Страховой калькулятора ОСАГО
8. Конвертер величин
9. Калькулятор металлопроката
10. Автомобильный шинный калькулятор
11. Калькулятор фундамента
12. Калькулятор ремонта квартиры
13. Калькулятор труб
14. Калькулятор калорий в продуктах
15. Калькулятор дат

Лабораторная работа №1. Задачи и класс Task

1. Цель и содержание

Цель лаботаторной работы: получить знания и навыки по работе с основным классом для параллельной работы.

2. Теоретическая часть

В эпоху многоядерных машин, которые позволяют параллельно выполнять сразу несколько процессов, стандартных средств работы с потоками в .NET уже оказалось недостаточно. Поэтому во фреймворк .NET была добавлена 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. Кроме того, упрощается сама работа по созданию новых потоков. Поэтому начиная с .NET 4.0. рекомендуется использовать именно TPL и ее классы для создания многопоточных приложений, хотя стандартные средства и класс Thread по-прежнему находят широкое применение.

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.

Для определения и запуска задачи можно использовать различные способы. Первый способ создание объекта Task и вызов у него метода Start:

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

В качестве параметра объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату, например, лямбда-выражение, как в данном случае, или ссылку на какой-

либо метод. То есть в данном случае при выполнении задачи на консоль будет выводиться строка "Hello Task!".

А метод Start() собственно запускает задачу.

Второй способ заключается в использовании статического метода Task.Factory.StartNew(). Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу:

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

В качестве результата метод возвращает запущенную задачу.

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

```
Task task = Task.Run(() => Console.WriteLine("Hello Task!"));
```

Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

Определим небольшую программу, где используем все эти способы:

```
using System;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task task1 = new Task(() => Console.WriteLine("Task1 is executed"));
            task1.Start();

            Task task2 = Task.Factory.StartNew(() => Console.WriteLine("Task2 is executed"));

            Task task3 = Task.Run(() => Console.WriteLine("Task3 is executed"));

            Console.ReadLine();
        }
    }
}
```

Важно понимать, что задачи не выполняются последовательно. Первая запущенная задача может завершенить свое выполнение после последней задачи.

Или рассмотрим еще один пример:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace TaskApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task task = new Task(Display);
            task.Start();

            Console.WriteLine("Завершение метода Main");

            Console.ReadLine();
        }

        static void Display()
        {
            Console.WriteLine("Начало работы метода Display");

            Console.WriteLine("Завершение работы метода Display");
        }
    }
}
```

Класс Task в качестве параметра принимает метод Display, который соответствует делегату Action. Далее чтобы запустить задачу, вызываем метод Start: task.Start(), и после этого метод Display начнет выполняться во вторичном потоке. В конце метода Main выводит некоторый маркер-строку, что метод Main завершился.

Однако в данном случае консольный вывод может выглядеть следующим образом:

```
Завершение метода Main
Начало работы метода Display
Завершение работы метода Display
```

То есть мы видим, что даже когда основной код в методе Main уже отработал, запущенная ранее задача еще не завершилась.

Чтобы указать, что метод Main должен подождать до конца выполнения задачи, нам надо использовать метод Wait:

```
static void Main(string[] args)
{
    Task task = new Task(Display);
    task.Start();
    task.Wait();
    Console.WriteLine("Завершение метода Main");
    Console.ReadLine();
}
```

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

- AsyncState: возвращает объект состояния задачи;
- CurrentId: возвращает идентификатор текущей задачи;
- Exception: возвращает объект исключения, возникшего при выполнении задачи;
- Status: возвращает статус задачи.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера,

установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.

2. Цели лабораторной работы.

3. Ответы на контрольные вопросы.

4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Библиотека, используемая C#;

2. Методы Task.

7. Задание

Рассмотреть полученный знания на примере в соответствии с выбранным вариантом.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №2. Работа с классом Task

1. Цель и содержание

Цель лаботаторной работы: Познакомиться с методами класса Task и научиться работать с некоторыми его методами.

2. Теоретическая часть

На прошлом занятии Вы познакомились с задачами класса Task. Одна задача может запускать другую - вложенную задачу. При этом эти задачи выполняются независимо друг от друга. Например:

```
static void Main(string[] args)
{
    var outer = Task.Factory.StartNew(() =>      // внешняя задача
    {
        Console.WriteLine("Outer task starting...");
        var inner = Task.Factory.StartNew(() =>  // вложенная задача
        {
            Console.WriteLine("Inner task starting...");
            Thread.Sleep(2000);
            Console.WriteLine("Inner task finished.");
        });
    });
    outer.Wait(); // ожидаем выполнения внешней задачи
    Console.WriteLine("End of Main");

    Console.ReadLine();
}
```

Несмотря на то, что здесь мы ожидаем выполнения внешней задачи, но вложенная задача может завершить выполнение даже после завершения метода Main:

```
Outer task starting...
End of Main
Inner task starting...
Inner task finished.
```

Если необходимо, чтобы вложенная задача выполнялась вместе с внешней, необходимо использовать значение TaskCreationOptions.AttachedToParent:

```
static void Main(string[] args)
{
    var outer = Task.Factory.StartNew(() =>      // внешняя задача
    {
        Console.WriteLine("Outer task starting...");
        var inner = Task.Factory.StartNew(() =>  // вложенная задача
        {
            Console.WriteLine("Inner task starting...");
            Thread.Sleep(2000);
            Console.WriteLine("Inner task finished.");
        }, TaskCreationOptions.AttachedToParent);
    });
    outer.Wait(); // ожидаем выполнения внешней задачи
    Console.WriteLine("End of Main");

    Console.ReadLine();
}
```

Консольный вывод:

```
Outer task starting...
Inner task starting...
Inner task finished.
End of Main
```

Также как и с потоками, мы можем создать и запустить массив задач. Можно определить все задачи в массиве непосредственно через объект Task:

```
Task[] tasks1 = new Task[3]
{
    new Task(() => Console.WriteLine("First Task")),
    new Task(() => Console.WriteLine("Second Task")),
    new Task(() => Console.WriteLine("Third Task"))
};
// запуск задач в массиве
foreach (var t in tasks1)
    t.Start();
```

Либо также можно использовать методы Task.Factory.StartNew или Task.Run и сразу запускать все задачи:

```
Task[] tasks2 = new Task[3];
int j = 1;
for (int i = 0; i < tasks2.Length; i++)
    tasks2[i] = Task.Factory.StartNew(() => Console.WriteLine($"Task {j++}"));
```

Но в любом случае мы опять же можем столкнуться, что все задачи из массива могут завершиться после того, как отработает метод Main, в котором запускаются эти задачи:

```
static void Main(string[] args)
{
    Task[] tasks1 = new Task[3]
    {
        new Task(() => Console.WriteLine("First Task")),
        new Task(() => Console.WriteLine("Second Task")),
        new Task(() => Console.WriteLine("Third Task"))
    };
    foreach (var t in tasks1)
        t.Start();

    Task[] tasks2 = new Task[3];
    int j = 1;
    for (int i = 0; i < tasks2.Length; i++)
        tasks2[i] = Task.Factory.StartNew(() => Console.WriteLine($"Task {j++}"));

    Console.WriteLine("Завершение метода Main");

    Console.ReadLine();
}
```

Однин из возможных консольных выводов программы:

```
Second Task
Task 1
Завершение метода Main
Third Task
Task 3
First Task
Task 2
```

Если необходимо выполнять некоторый код лишь после того, как все задачи из массива завершатся, то применяется метод Task.WaitAll(tasks):

```
static void Main(string[] args)
{
    Task[] tasks1 = new Task[3]
    {
        new Task(() => Console.WriteLine("First Task")),
        new Task(() => Console.WriteLine("Second Task")),
        new Task(() => Console.WriteLine("Third Task"))
    };
    foreach (var t in tasks1)
        t.Start();
    Task.WaitAll(tasks1); // ожидаем завершения задач

    Console.WriteLine("Завершение метода Main");

    Console.ReadLine();
}
```

В этом случае сначала завершатся все задачи, и лишь только потом будет выполняться последующий код из метода Main:

```
Second Task
Third Task
First Task
Завершение метода Main
```

В то же время порядок выполнения самих задач в массиве также недетерминирован.

Также мы можем применять метод Task.WaitAny(tasks). Он ждет, пока завершится хотя бы одна из массива задач.

Возвращение результатов из задач

Задачи могут не только выполняться как процедуры, но и возвращать определенные результаты:

```
class Program
{
    static void Main(string[] args)
    {
        Task<int> task1 = new Task<int>(()=>Factorial(5));
        task1.Start();

        Console.WriteLine($"Факториал числа 5 равен {task1.Result}");

        Task<Book> task2 = new Task<Book>(() =>
        {
            return new Book { Title = "Война и мир", Author = "Л. Толстой" };
        });
        task2.Start();

        Book b = task2.Result;  // ожидаем получение результата
        Console.WriteLine($"Название книги: {b.Title}, автор: {b.Author}");

        Console.ReadLine();
    }
```

```
    static int Factorial(int x)
    {
        int result = 1;

        for (int i = 1; i <= x; i++)
        {
            result *= i;
        }

        return result;
    }
}

public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
}
```

Во-первых, чтобы задать возвращаемый из задачи тип объекта, мы должны типизировать Task. Например, Task<int> - в данном случае задача будет возвращать объект int.

И, во-вторых, в качестве задачи должен выполняться метод, возвращающий данный тип объекта. Например, в первом случае у нас в качестве задачи выполняется функция Factorial, которая принимает числовой параметр и также на выходе возвращает число.

Возвращаемое число будет храниться в свойстве Result: task1.Result. Нам не надо его приводить к типу int, оно уже само по себе будет представлять число.

То же самое и со второй задачей task2. В этом случае в лямбда-выражении возвращается объект Book. И также мы его получаем с помощью task2.Result

При этом при обращении к свойству Result программа текущий поток останавливает выполнение и ждет, когда будет получен результат из выполняемой задачи.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя

персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.

2. Цели лабораторной работы.

3. Ответы на контрольные вопросы.

4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Лямбда-выражения.

2. Массив задач.

3. Вложенные задачи.

7. Задание

Добавить примеры Task в соответствии с вариантом.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №3. Задачи продолжения

1. Цель и содержание

Цель лаботаторной работы: Рассмотреть примеры задач продолжения

2. Теоретическая часть

Задачи продолжения или continuation task позволяют определить задачи, которые выполняются после завершения других задач. Благодаря этому мы можем вызвать после выполнения одной задачи несколько других, определить условия их вызова, передать из предыдущей задачи в следующую некоторые данные.

Задачи продолжения похожи на методы обратного вызова, но фактически являются обычными задачами Task. Посмотрим на примере:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace TaskApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task task1 = new Task(()=>{
                    Console.WriteLine($"Id задачи: {Task.CurrentId}");
                });

            // задача продолжения
            Task task2 = task1.ContinueWith(Display);

            task1.Start();

            // ждем окончания второй задачи
            task2.Wait();
            Console.WriteLine("Выполняется работа метода Main");
            Console.ReadLine();
        }

        static void Display(Task t)
        {
            Console.WriteLine($"Id задачи: {Task.CurrentId}");
            Console.WriteLine($"Id предыдущей задачи: {t.Id}");
            Thread.Sleep(3000);
        }
    }
}
```

Первая задача задается с помощью лямбда-выражения, которое просто выводит id этой задачи. Вторая задача - задача продолжения задачется с помощью метода ContinueWith, который в качестве параметра принимает делегат Action<Task>. То есть метод Display, который передается в данный метод в качестве значения параметра, должен принимать параметр типа Task.

Благодаря передачи в метод параметра Task, мы можем получить различные свойства предыдущей задачи, как например, в данном случае получает ее Id.

И после завершения задачи task1 сразу будет вызываться задача task2.

Также мы можем передавать конкретный результат работы предыдущей задачи:

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;
using System.Threading;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Task<int> task1 = new Task<int>(()=>Sum(4,5));

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

            task1.Start();

            // ждем окончания второй задачи
            task2.Wait();
            Console.WriteLine("End of Main");
            Console.ReadLine();
        }

        static int Sum(int a, int b) => a + b;
        static void Display(int sum)
        {
            Console.WriteLine($"Sum: {sum}");
        }

    }
}
</int>
```

Подобным образом можно построить целую цепочку последовательно выполняющихся задач:

```
static void Main(string[] args)
{
    Task task1 = new Task(()=>{
        Console.WriteLine($"Id задачи: {Task.CurrentId}");
    });

    // задача продолжения
    Task task2 = task1.ContinueWith(Display);

    Task task3 = task1.ContinueWith((Task t) =>
    {
        Console.WriteLine($"Id задачи: {Task.CurrentId}");
    });

    Task task4 = task2.ContinueWith((Task t) =>
    {
        Console.WriteLine($"Id задачи: {Task.CurrentId}");
    });

    task1.Start();

    Console.ReadLine();
}
```

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера,

установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.

2. Цели лабораторной работы.

3. Ответы на контрольные вопросы.

4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Задача продолжения.

2. Делегаты

7. Задание

Расширить существующее приложение задачами с продолжением. Выбранная задача не должна идти в разрез с основной задачей приложения.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №4. Класс Parallel

1. Цель и содержание

Цель лаботаторной работы: Познакомиться с классом Parallel.

2. Теоретическая часть

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:

```
static void Main(string[] args)
{
    Parallel.Invoke(Display,
        () => {
                    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
                    Thread.Sleep(3000);
                },
        () => Factorial(5));

    Console.ReadLine();
}

static void Display()
{
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Thread.Sleep(3000);
}

static void Factorial(int x)
{
    int result = 1;

    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Thread.Sleep(3000);
    Console.WriteLine($"Результат {result}");
}
```

Метод Parallel.Invoke в качестве параметра принимает массив объектов Action, то есть мы можем передать в данный метод набор методов, которые

будут вызываться при его выполнении. Количество методов может быть различным, но в данном случае мы определяем выполнение трех методов. Опять же как и в случае с классом Task мы можем передать либо название метода, либо лямбда-выражение.

И таким образом, при наличии нескольких ядер на целевой машине данные методы будут выполняться параллельно на различных ядрах.

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:

```
static void Main(string[] args)
{
    Parallel.For(1, 10, Factorial);

    Console.ReadLine();
}

static void Factorial(int x)
{
    int result = 1;

    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Console.WriteLine($"Факториал числа {x} равен {result}");
    Thread.Sleep(3000);
}
```

В данном случае в качестве первого параметра в метод Parallel.For передается число 1, а в качестве второго - число 10. Таким образом, метод будет вести итерацию с 1 до 9 включительно. Третий параметр представляет метод, подсчитывающий факториал числа. Так как этот параметр представляет тип Action<int>, то этот метод в качестве параметра должен принимать объект int.

А в качестве значения параметра в этот метод передается счетчик, который проходит в цикле от 1 до 9 включительно. И метод Factorial, таким образом, вызывается 9 раз.

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, которая содержит информацию о выполнении цикла.

```
static void Main(string[] args)
{
    ParallelLoopResult result = Parallel.ForEach<int>(new List<int>() { 1, 3, 5, 8 },
                Factorial);

    Console.ReadLine();
}
static void Factorial(int x)
{
    int result = 1;

    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Console.WriteLine($"Факториал числа {x} равен {result}");
    Thread.Sleep(3000);
}
```

В данном случае поскольку мы используем коллекцию объектов int, то и метод, который мы передаем в качестве второго параметра, должен в качестве параметра принимать значение int.

В стандартных циклах for и foreach предусмотрен преждевременный выход из цикла с помощью оператора break. В методах Parallel.ForEach и Parallel.For мы также можем, не дожидаясь окончания цикла, выйти из него:

```
static void Main(string[] args)
{
    ParallelLoopResult result = Parallel.For(1, 10, Factorial);

    if (!result.IsCompleted)
        Console.WriteLine($"Выполнение цикла завершено на итерации {result.LowestBreakIteration}");
    Console.ReadLine();
}
static void Factorial(int x, ParallelLoopState pls)
{
    int result = 1;

    for (int i = 1; i <= x; i++)
    {
        result *= i;
        if (i == 5)
            pls.Break();
    }
    Console.WriteLine($"Выполняется задача {Task.CurrentId}");
    Console.WriteLine($"Факториал числа {x} равен {result}");
}
```

Здесь метод Factorial, обрабатывающий каждую итерацию, принимает дополнительный параметр - объект ParallelLoopState. И если счетчик в цикле достигнет значения 5, вызывается метод Break. Благодаря чему система осуществит выход и прекратит выполнение метода Parallel.For при первом удобном случае.

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

Так как у нас на индексе равном 5 происходит прерывание, то свойство IsCompleted будет иметь значение false, а LowestBreakIteration будет равно 5.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше,

оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Invoke.
2. Циклы.
3. Выход из циклов.

7. Задание

Добавить использование класса Parallel в своё приложение.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №5. Отмена задач и параллельных операций

1. Цель и содержание

Цель лаботаторной работы: Научиться работать с CancellationToken.

2. Теоретическая часть

Параллельное выполнение задач может занимать много времени. И иногда может возникнуть необходимость прервать выполняемую задачу. Для этого .NET предоставляет класс CancellationToken:

```
static void Main(string[] args)
{
    CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
    CancellationToken token = cancelTokenSource.Token;
    int number = 6;

    Task task1 = new Task(() =>
    {
        int result = 1;
        for (int i = 1; i <= number; i++)
        {
            if (token.IsCancellationRequested)
            {
                Console.WriteLine("Операция прервана");
                return;
            }

            result *= i;
            Console.WriteLine($"Факториал числа {number} равен {result}");
            Thread.Sleep(5000);
        }
    });
    task1.Start();

    Console.WriteLine("Введите Y для отмены операции или другой символ для ее продолжения:");
    string s = Console.ReadLine();
    if (s == "Y")
        cancelTokenSource.Cancel();

    Console.Read();
}
```

Для отмены операции нам надо создать и использовать токен. Вначале создается объект CancellationTokenSource:

```
CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
```

Затем из него получаем сам токен:

```
CancellationToken token = cancelTokenSource.Token;
```

Чтобы отменить операцию, необходимо вызвать метод Cancel() у объекта CancellationTokenSource:

```
cancelTokenSource.Cancel();
```

В самой операции мы можем отловить выставление токена с помощью условной конструкции:

```
if (token.IsCancellationRequested)
{
    Console.WriteLine("Операция прервана токеном");
    return;
}
```

Если был вызван метод cancelTokenSource.Cancel(), то выражение token.IsCancellationRequested возвращает true.

Если операция представляет внешний метод, то ему надо передавать в качестве одного из параметров токен:

```
static void Main(string[] args)
{
    CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
    CancellationToken token = cancelTokenSource.Token;

    Task task1 = new Task(() => Factorial(5, token));
    task1.Start();

    Console.WriteLine("Введите Y для отмены операции или любой другой символ для ее продолжения:");
    string s = Console.ReadLine();
    if (s == "Y")
        cancelTokenSource.Cancel();

    Console.ReadLine();
}

static void Factorial(int x, CancellationToken token)
{
    int result = 1;
    for (int i = 1; i <= x; i++)
    {
        if (token.IsCancellationRequested)
        {
            Console.WriteLine("Операция прервана токеном");
            return;
        }

        result *= i;
        Console.WriteLine($"Факториал числа {x} равен {result}");
        Thread.Sleep(5000);
    }
}
```

Для отмены выполнения параллельных операций, запущенных с помощью методов Parallel.For() и Parallel.ForEach(), можно использовать перегруженные версии данных методов, которые принимают в качестве параметра объект ParallelOptions. Данный объект позволяет установить токен:

```
static void Main(string[] args)
{
    CancellationTokenSource cancelTokenSource = new CancellationTokenSource();
    CancellationToken token = cancelTokenSource.Token;

    new Task(()=>
    {
        Thread.Sleep(400);
        cancelTokenSource.Cancel();
    }).Start();

    try
    {
        Parallel.ForEach<int>(new List<int>() { 1,2,3,4,5,6,7,8},
                              new ParallelOptions { CancellationToken=token}, Factorial);
        // или так
        //Parallel.For(1, 8, new ParallelOptions { CancellationToken = token }, Factorial);
    }
    catch(OperationCanceledException ex)
    {
        Console.WriteLine("Операция прервана");
    }
    finally
    {
        cancelTokenSource.Dispose();
    }

    Console.ReadLine();
}
static void Factorial(int x)
{
    int result = 1;

    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    Console.WriteLine($"Факториал числа {x} равен {result}");
    Thread.Sleep(3000);
}
```

В параллельной запущеной задаче через 400 миллисекунд происходит вызов cancelTokenSource.Cancel(), в результате программа выбрасывает

исключение OperationCanceledException, и выполнение параллельных операций прекращается.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.

2. Цели лабораторной работы.

3. Ответы на контрольные вопросы.

4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Класс CancellationToken.

2. Основные методы класса CancellationToken.

7. Задание

Расширить Ваше приложение задачами, которые можно отменять.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №6. Асинхронные методы, async и await

1. Цель и содержание

Цель лаботаторной работы: Рассмотреть асинхронность выполнения с помощью ключевых слов async и await.

2. Теоретическая часть

Асинхронность позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы или блоки кода. Особенно это актуально в графических программах, где продолжительные задачи могу блокировать интерфейс пользователя. И чтобы этого не произошло, нужно задействовать асинхронность. Также асинхронность несет выгоды в веб-приложениях при обработке запросов от пользователей, при обращении к базам данных или сетевым ресурсам. При больших запросах к базе данных асинхронный метод просто уснет на время, пока не получит данные от БД, а основной поток сможет продолжить свою работу. В синхронном же приложении, если бы код получения данных находился в основном потоке, этот поток просто бы блокировался на время получения данных.

Ключевыми для работы с асинхронными вызовами в C# являются два ключевых слова: async и await, цель которых - упростить написание асинхронного кода. Они используются вместе для создания асинхронного 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.

Рассмотрим пример асинхронного метода:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static void Factorial()
        {
            int result = 1;
            for(int i = 1; i <= 6; i++)
            {
                result *= i;
            }
            Thread.Sleep(8000);
            Console.WriteLine($"Факториал равен {result}");
        }
        // определение асинхронного метода
        static async void FactorialAsync()
        {
            Console.WriteLine("Начало метода FactorialAsync"); // выполняется синхронно
            await Task.Run(()=>Factorial());                   // выполняется асинхронно
            Console.WriteLine("Конец метода FactorialAsync");
        }

        static void Main(string[] args)
        {
            FactorialAsync();   // вызов асинхронного метода

            Console.WriteLine("Введите число: ");
            int n = Int32.Parse(Console.ReadLine());
            Console.WriteLine($"Квадрат числа равен {n * n}");

            Console.Read();
        }
    }
}
```

Здесь прежде всего определен обычный метод подсчета факториала. Для имитации долгой работы в нем используется задержка на 8 секунд с помощью метода Thread.Sleep(). Условно это некоторый метод, который выполняет некоторую работу продолжительное время. Но для упрощения понимания он просто подсчитывает факториал числа 6.

Также здесь определен асинхронный метод FactorialAsync(). Асинхронным он является потому, что имеет в определении перед возвращаемым типом модификатор async, его возвращаемым типом является void, и в теле метода определено выражение await.

Выражение await определяет задачу, которая будет выполняться асинхронно. В данном случае подобная задача представляет выполнение функции факториала:

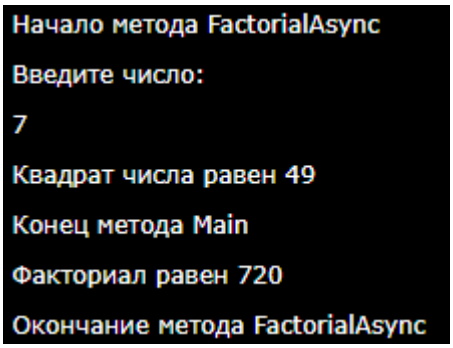
```
await Task.Run(()=>Factorial());
```

По негласным правилам в названии асинхроннных методов принято использовать суффикс Async - FactorialAsync(), хотя в принципе это необязательно делать.

Сам факториал мы получаем в асинхронном методе FactorialAsync. Асинхронным он является потому, что он объявлен с модификатором async и содержит использование ключевого слова await.

И в методе Main мы вызываем этот асинхронный метод.

Посмотрим, какой у программы будет консольный вывод:



Разберем поэтапно, что здесь происходит:

1. Запускается метод Main, в котором вызывается асинхронный метод FactorialAsync.

2. Метод FactorialAsync начинает выполняться синхронно вплоть до выражения await.

3. Выражение await запускает асинхронную задачу Task.Run(()=>Factorial())

4. Пока выполняется асинхронная задача Task.Run(()=>Factorial()) (а она может выполняться довольно продожительное время), выполнение кода возвращается в вызывающий метод - то есть в метод Main. В методе Main нам будет предложено ввести число для вычисления квадрата числа.

В этом и преимущество асинхронных методов - асинхронная задача, которая может выполняться довольно долгое время, не блокирует метод Main, и мы можем продолжать работу с ним, например, вводить и обрабатывать данные.

5. Когда асинхронная задача завершила свое выполнение (в случае выше - подсчитала факториал числа), продолжает работу асинхронный метод FactorialAsync, который вызвал асинхронную задачу.

Функция факториала, возможно, представляет не самый показательный пример, так как в реальности в данном случае нет смысла делать ее асинхронной. Но рассмотрим другой пример - чтение-запись файла:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.IO;

namespace HelloApp
{
    class Program
    {
        static async void ReadWriteAsync()
        {
            string s = "Hello world! One step at a time";

            // hello.txt - файл, который будет записываться и считываться
            using (StreamWriter writer = new StreamWriter("hello.txt", false))
            {
                await writer.WriteLineAsync(s);  // асинхронная запись в файл
            }
            using (StreamReader reader = new StreamReader("hello.txt"))
            {
                string result = await reader.ReadToEndAsync();  // асинхронное чтение из файла
                Console.WriteLine(result);
            }
        }
        static void Main(string[] args)
        {
            ReadWriteAsync();

            Console.WriteLine("Некоторая работа");
            Console.Read();
        }
    }
}
```

Асинхронный метод ReadWriteAsync() выполнение запись в файл некоторой строки и затем считывает записанный файл. Подобные операции могут занимать продолжительное время, особенно при больших объемах данных, поэтому такие операции лучше делать асинхронными.

Фреймворк .NET уже имеет встроенную поддержку таких операций. Например, в классе StreamWriter определен метод WriteLineAsync(). По сути он уже представляет асинхронную операцию и принимает в качестве параметра некоторую строку, которую надо записать в файл. Поскольку этот метод представляет асинхронную операцию, то вызов этого метода мы можем оформить в выражение await:

```
await writer.WriteLineAsync(s);
```

Аналогично в классе StreamReader определен метод ReadToEndAsync(), который также представляет асинхронную операцию и который возвращает весь считанный текст.

Во фреймворке .NET Core определено много подобных методов. Как правило, они связаны с работой с файлами, отправкой сетевых запросов или запросов к базе данных. Их легко узнать по суффиксу Async. То есть если метод имеет подобный суффикс в названии, то с большей степенью вроятности его можно использовать в выражении await.

Далее в методе Main вызывается асинхронный метод ReadWriteAsync:

```
static void Main(string[] args)
{
    ReadWriteAsync();

    Console.WriteLine("Некоторая работа");
    Console.Read();
}
```

И опять же когда выполнение в методе ReadWriteAsync доходит до первого выражения await, управление возвращается в метод Main, и мы можем продолжать с ним работу. Запись в файл и считывания файла будут производиться параллельно и не будут блокировать работу метода Main.

Как, выше уже было сказано, фреймворк .NET Core имеет много встроенных методов, которые представляют асинхронную операцию. Они заканчиваются на суффикс Async. И перед вызывами подобных методов мы можем указывать оператор await. Например:

```
StreamWriter writer = new StreamWriter("hello.txt", false);
await writer.WriteLineAsync("Hello");
```

Либо мы сами можем определить асинхронную операцию, используя метод Task.Run():

```
static void Factorial()
{
    int result = 1;
    for (int i = 1; i <= 6; i++)
    {
        result *= i;
    }
    Thread.Sleep(8000);
    Console.WriteLine($"Факториал равен {result}");
}
// определение асинхронного метода
static async void FactorialAsync()
{
    await Task.Run(()=>Factorial()); // вызов асинхронной операции
}
```

Можно определить асинхронную операцию с помощью лямбда-выражения:

```
static async void FactorialAsync()
{
    await Task.Run(() =>
    {
        int result = 1;
        for (int i = 1; i <= 6; i++)
        {
            result *= i;
        }
        Thread.Sleep(8000);
        Console.WriteLine($"Факториал равен {result}");
    });
}
```

Выше вычислялся факториал 6, но, допустим, мы хотим вычислять факториалы разных чисел:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static void Factorial(int n)
        {
            int result = 1;
            for (int i = 1; i <= n; i++)
            {
                result *= i;
            }
            Thread.Sleep(5000);
            Console.WriteLine($"Факториал равен {result}");
        }
        // определение асинхронного метода
        static async void FactorialAsync(int n)
        {
            await Task.Run(()=>Factorial(n));
        }
        static void Main(string[] args)
        {
            FactorialAsync(5);
            FactorialAsync(6);
            Console.WriteLine("Некоторая работа");
            Console.Read();
        }
    }
}
```

Асинхронная операция может возвращать некоторый результат, получить который мы можем так же, как и при вызове обычного метода:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static int Factorial(int n)
        {
            int result = 1;
            for (int i = 1; i <= n; i++)
            {
                result *= i;
            }
            return result;
        }
        // определение асинхронного метода
        static async void FactorialAsync(int n)
        {
            int x = await Task.Run(()=>Factorial(n));
            Console.WriteLine($"Факториал равен {x}");
        }
        static void Main(string[] args)
        {
            FactorialAsync(5);
            FactorialAsync(6);
            Console.Read();
        }
    }
}
```

Метод Factorial возвращает значение типа int, это значение мы можем получить, просто присвоив результат асинхронной операции переменной данного типа: int x = await Task.Run(()=>Factorial(n));

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение:

операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Асинхронность.
2. Определение асинхронной операции.
3. Передача парамметров в асинхронную операцию.
4. Получение результата из асинхронной операции.

7. Задание

Изменить ранее созданные методы методами с использованием async и await.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №7. Возвращение результата из асинхронного метода

1. Цель и содержание

Цель лаботаторной работы: Научиться работать с результатом выполнения асинхронной задачи.

2. Теоретическая часть

В качестве возвращаемого типа в асинхронном методе должны использоваться типы void, Task, Task<T> или ValueTask<T>.

1. void

При использовании ключевого слова void асинхронный метод ничего не возвращает:

```
static void Factorial(int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }
    Console.WriteLine($"Факториал равен {result}");
}
// определение асинхронного метода
static async void FactorialAsync(int n)
{
    await Task.Run(()=>Factorial(n));
}
```

2. Task

Возвращение объекта типа Task:

```
using System;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static void Factorial(int n)
        {
            int result = 1;
            for (int i = 1; i <= n; i++)
            {
                result *= i;
            }
            Console.WriteLine($"Факториал равен {result}");
        }

        // определение асинхронного метода
        static async Task FactorialAsync(int n)
        {
            await Task.Run(()=>Factorial(n));
        }
        static void Main(string[] args)
        {
            FactorialAsync(5);
            FactorialAsync(6);
            Console.WriteLine("Некоторая работа");
            Console.Read();
        }
    }
}
```

Формально метод FactorialAsync не использует оператор return для возвращения результата. Однако если в асинхронном методе выполняется в выражении await асинхронная операция, то мы можем возвращать из метода объект Task.

3. Task<T>

Метод может возвращать некоторое значение. Тогда возвращаемое значение оборачивается в объект Task, а возвращаемым типом является Task<T>:

```
using System;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static int Factorial(int n)
        {
            int result = 1;
            for (int i = 1; i <= n; i++)
            {
                result *= i;
            }
            return result;
        }
        // определение асинхронного метода
        static async Task<int> FactorialAsync(int n)
        {
            return await Task.Run(()=>Factorial(n));
        }
        static async Task Main(string[] args)
        {
            int n1 = await FactorialAsync(5);
            int n2 = await FactorialAsync(6);
            Console.WriteLine($"n1={n1}  n2={n2}");
            Console.Read();
        }
    }
}
```

В данном случае функция Factorial возвращает значение типа int. В асинхронном методе FactorialAsync мы получаем и возвращаем это число. Поэтому возвращаемым типом в данном случае является типа Task<int>. Если бы метод Factorial возвращал строку, то есть данные типа string, то возвращаемым типом асинхронного метода был бы тип Task<string>

Чтобы получить результат асинхронного метода в методе Main, который тоже определен как асинхронный, применяем оператор await при вызове FactorialAsync.

4. ValueTask<T>

Использование типа ValueTask<T> во многом аналогично применению Task<T> за исключением некоторых различий в работе с памятью, поскольку ValueTask - структура, а Task - класс. По умолчанию тип ValueTask

недоступен, и чтобы использовать его, вначале надо установить через NuGet пакет System.Threading.Tasks.Extensions.

Использование ValueTask:

```
using System;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static int Factorial(int n)
        {
            int result = 1;
            for (int i = 1; i <= n; i++)
            {
                result *= i;
            }
            return result;
        }
        // определение асинхронного метода
        static async ValueTask<int> FactorialAsync(int n)
        {
            return await Task.Run(()=>Factorial(n));
        }
        static async Task Main(string[] args)
        {
            int n1 = await FactorialAsync(5);
            int n2 = await FactorialAsync(6);
            Console.WriteLine($"n1={n1}  n2={n2}");
            Console.Read();
        }
    }
}
```

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Возвращаемый тип void.
2. Возвращаемый тип Task.
3. Возвращаемый тип Task<T>.
4. Возвращаемый типValueTask<T>.

7. Задание

Полученный в предыдущей лабораторной работе результат отобразить в консоли.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №8. Последовательный и параллельный вызов асинхронных операций.

1. Цель и содержание

Цель лаботаторной работы: Научиться работать с несколькими вызовами await.

2. Теоретическая часть

Асинхронный метод может содержать множество выражений await. Когда система встречает в блоке кода оператор await, то выполнение в асинхронном методе останавливается, пока не завершится асинхронная задача. После завершения задачи управление переходит к следующему оператору await и так далее. Это позволяет вызывать асинхронные задачи последовательно в определенном порядке. Например:

```
using System;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static void Factorial(int n)
        {
            int result = 1;
            for (int i = 1; i <= n; i++)
            {
                result *= i;
            }
            Console.WriteLine($"Факториал числа {n} равен {result}");
        }
        // определение асинхронного метода
        static async void FactorialAsync()
        {
            await Task.Run(() => Factorial(4));
            await Task.Run(() => Factorial(3));
            await Task.Run(() => Factorial(5));
        }
        static void Main(string[] args)
        {
            FactorialAsync();
            Console.Read();
        }
    }
}
```

Консольный вывод данной программы:

```
Факториал числа 4 равен 24
Факториал числа 3 равен 6
Факториал числа 5 равен 120
```

То есть мы видим, что факториалы вычисляются последовательно. И в данном случае вывод строго детерминирован.

Нередко такая последовательность бывает необходима, если одна задача зависит от результатов другой.

Однако не всегда существует подобная зависимость между задачами. В этом случае мы можем запустить все задачи параллельно и через метод Task.WhenAll отследить их завершение. Например, изменим метод FactorialAsync:

```
static async void FactorialAsync()
{
    Task t1 = Task.Run(() => Factorial(4));
    Task t2 = Task.Run(() => Factorial(3));
    Task t3 = Task.Run(() => Factorial(5));
    await Task.WhenAll(new[] { t1, t2, t3 });
}
```

Вначале запускаются три задачи. Затем Task.WhenAll создает новую задачу, которая будет автоматически выполнена после выполнения всех предоставленных задач, то есть задач t1, t2, t3. А с помощью оператора await ожидаем ее завершения.

В итоге все три задачи теперь будут запускаться параллельно, однако вызывающий метод FactorialAsync благодаря оператору await все равно будет ожидать, пока они все не завершатся. И в этом случае вывод программы не детерминирован. Например, он может быть следующим:

```
Факториал числа 5 равен 120
Факториал числа 4 равен 24
Факториал числа 3 равен 6
```

И если задача возвращает какое-нибудь значение, то это значение потом можно получить с помощью свойства Result.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.

2. Цели лабораторной работы.

3. Ответы на контрольные вопросы.

4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1.Метод WhenAll.

2. Свойство Result.

7. Задание

Использованны ранее методы разбить на две группы: те, которые будут выполняться последовательно и те, которые выполняются параллельною

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №9. Обработка ошибок в асинхронных методах

1. Цель и содержание

Цель лабототаторной работы: Рассмотреть методы обработки ошибок асинхронных операций.

2. Теоретическая часть

Обработка ошибок в асинхронных методах, использующих ключевые слова async и await, имеет свои особенности.

Для обработки ошибок выражение await помещается в блок try:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static void Factorial(int n)
        {
            if (n < 1)
                throw new Exception($"{n} : число не должно быть меньше 1");
            int result = 1;
            for (int i = 1; i <= n; i++)
            {
                result *= i;
            }
            Console.WriteLine($"Факториал числа {n} равен {result}");
        }

        static async void FactorialAsync(int n)
        {
            try
            {
                await Task.Run(() => Factorial(n));
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }

        static void Main(string[] args)
        {
            FactorialAsync(-4);
            FactorialAsync(6);

            Console.Read();
        }
    }
}
```

В данном случае метод Factorial генерирует исключение, если методу передается число меньше 0.

Для обработки исключения в методе FactorialAsync выражение await помещено в блок try.

В методе Main вызывается асинхронный метод с передачей ему отрицательного числа: FactorialAsync(-4), что привет к генерации исключения. Однако программа не остановит аварийно свою работу, а обработает исключение и продолжит дальнейшие вычисления.

Следует учитывать, что если мы запускаем данный код в режиме отладки в Visual Studio, то VS просигнализирует нам о генерации исключении и остановит выполнение а строке throw new Exception($"{n} : число не должно быть меньше 1");. В режиме запуска без отладки VS не будет останавливаться.

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. Чтобы проинспектировать свойство, изменим метод FactorialAsync следующим образом:

```
static async void FactorialAsync(int n)
{
    Task task = null;
    try
    {
        task = Task.Run(()=>Factorial(n));
        await task;
    }
    catch (Exception ex)
    {
        Console.WriteLine(task.Exception.InnerException.Message);
        Console.WriteLine($"IsFaulted: {task.IsFaulted}");
    }
}
```

И если мы передадим в метод число -1, то task.IsFaulted будет равно true.

Обработка нескольких исключений. WhenAll

Если мы ожидаем выполнения сразу нескольких задач, например, с помощью Task.WhenAll, то мы можем получить сразу несколько исключений одномоментно для каждой выполняемой задачи. В этом случае мы можем получить все исключения из свойства Exception.InnerExceptions:

```
static async Task DoMultipleAsync()
{
    Task allTasks = null;

    try
    {
        Task t1 = Task.Run(()=>Factorial(-3));
        Task t2 = Task.Run(() => Factorial(-5));
        Task t3 = Task.Run(() => Factorial(-10));

        allTasks = Task.WhenAll(t1, t2, t3);
        await allTasks;
    }
    catch (Exception ex)
    {
        Console.WriteLine("Исключение: " + ex.Message);
        Console.WriteLine("IsFaulted: " + allTasks.IsFaulted);
        foreach (var inx in allTasks.Exception.InnerExceptions)
        {
            Console.WriteLine("Внутреннее исключение: " + inx.Message);
        }
    }
}
```

Здесь в три вызова метода факториала передаются заведомо некорректные числа: -3, -5, -10. Таким образом, при всех трх вызовах будет сгенерирована ошибка.

Хотя блок catch через переменную Exception ex будет получать одно перехваченное исключение, но с помощью коллекции Exception.InnerExceptions мы сможем получить инфрмацию обо всех возникших исключениях.

В итоге при выполнении этого метода мы получим следующий консольный вывод:

```
Исключение: -3 : число не должно быть меньше 1
IsFaulted: True
Внутреннее исключение: -3: число не должно быть меньше 1
Внутреннее исключение: -5: число не должно быть меньше 1
Внутреннее исключение: -10: число не должно быть меньше 1
```

Начиная с версии C# 6.0 в язык была добавлена возможность вызова асинхронного кода в блоках catch и finally. Так, возьмем предыдущий пример с подсчетом факториала:

```
static async void FactorialAsync(int n)
{
    try
    {
        await Task.Run(() => Factorial(n)); ;
    }
    catch (Exception ex)
    {
        await Task.Run(()=>Console.WriteLine(ex.Message));
    }
    finally
    {
        await Task.Run(() => Console.WriteLine("await в блоке finally"));
    }
}
```

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности

персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.

2. Цели лабораторной работы.

3. Ответы на контрольные вопросы.

4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Блок Try.

2. Исследование исключения.

3. Обработка нескольких исключений.

7. Задание

Использовать полученные навыки для добавления обработки ошибок в одном из методов.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №10. Отмена асинхронных операций

1. Цель и содержание

Цель лаботаторной работы: Научиться отменять асинхронные операции.

2. Теоретическая часть

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.

Рассмотрим применение этих классов на примере:

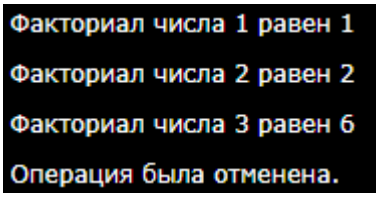
```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static void Factorial(int n, CancellationToken token)
        {
            int result = 1;
            for (int i = 1; i <= n; i++)
            {
                if (token.IsCancellationRequested)
                {
                    Console.WriteLine("Операция прервана токеном");
                    return;
                }
                result *= i;
                Console.WriteLine($"Факториал числа {i} равен {result}");
                Thread.Sleep(1000);
            }
        }
        // определение асинхронного метода
        static async void FactorialAsync(int n, CancellationToken token)
        {
            if(token.IsCancellationRequested)
                return;
            await Task.Run(()=>Factorial(n, token));
        }

        static void Main(string[] args)
        {
            CancellationTokenSource cts = new CancellationTokenSource();
            CancellationToken token = cts.Token;
            FactorialAsync(6, token);
            Thread.Sleep(3000);
            cts.Cancel();
            Console.Read();
        }
    }
}
```

Для создания токена определяется объект CancellationTokenSource. Метод FactorialAsync в качестве параметра принимает токен, и если где-то во внешнем коде произойдет отмена операции через вызов cts.Cancel, то в методе Factorial свойство token.IsCancellationRequested будет равно true, и соответственно при очередной итерации цикла в методе Factorial произойдет

выход из метода. И асинхронная операция завершится. И мы получим следующий консольный вывод:



3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.

3. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Задание

Создать ещё один асинхронный метод, который можно будет отменить по нажатию на любую клавишу. После отмены отобразить промежуточный результат в консоли.

7. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) поясняет ход выполнения индивидуального задания;

2) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №11. Асинхронные стримы

1. Цель и содержание

Цель лаботаторной работы: Рассмотреть асинхронные потоки и научиться с ними работать.

2. Теоретическая часть

Начиная с версии C# 8.0 в C# были добавлены асинхронные стримы, которые упрощают работу со потоками данных в асинхронном режиме. Хотя асинхронность в C# существует уже довольно давно, тем не менее асинхронные методы до сих пор позволяли получать один объект, когда асинхронная операция была готова предоставить результат. Для возвращения нескольких значений в C# могут применяться итераторы, но они имеют синхронную природу, блокируют вызывающий поток и не могут использоваться в асинхронном контексте. Асинхронные стримы обходят эту проблему, позволяя получать множество значений и возвращать их по мере готовности в асинхронном режиме.

По сути асинхронный стрим представляет метод, который обладает тремя характеристиками:

- метод имеет модификатор async
- метод возвращает объект IAsyncEnumerable<T>. Интерфейс IAsyncEnumerable определяет метод GetAsyncEnumerator, который возвращает IAsyncEnumerator:

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken cancellationToken = default
}

public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    T Current { get; }
    ValueTask<bool> MoveNextAsync();
}
public interface IAsyncDisposable
{
    ValueTask DisposeAsync();
}
```

- метод содержит выражения yield return для последовательного получения элементов из асинхронного стрима.

Фактически асинхронный стрим объединяет асинхронность и итераторы. Рассмотрим простейший пример:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace HelloApp
{
    class Program
    {
        static async Task Main(string[] args)
        {
            await foreach (var number in GetNumbersAsync())
            {
                Console.WriteLine(number);
            }
        }

        public static async IAsyncEnumerable<int> GetNumbersAsync()
        {
            for (int i = 0; i < 10; i++)
            {
                await Task.Delay(100);
                yield return i;
            }
        }
    }
}
```

Итак, метод GetNumbersAsync() фактически и представляет асинхронный стрим. Этот метод является асинхронным. Его возвращаемый тип - IAsyncEnumerable<int>. А его суть сводится к тому, что он возвращает с помощью yield return каждый 100 некоторое число. То есть фактически метод должен вернуть 10 чисел от 0 до 10 с промежутком в 100 миллисекунд.

Для получения данных из стрима в методе Main используется цикл foreach:

```
await foreach (var number in GetNumbersAsync())
```

Важно отметить, что он предваряется оператором await. И в этом случае метод Main должен быть определен с оператором async:

```
static async Task Main(string[] args)
```

В итоге, каждый раз когда асинхронный стрим будет возвращать очередное число, цикл будет его получать и выводить на консоль.

Где можно применять асинхронные стримы? Асинхронные стримы могут применяться для получения данных из какого-нибудь внешнего хранилища. Например, пусть имеется следующий класс некоторого хранилища:

```
class Repository
{
    string[] data = { "Tom", "Sam", "Kate", "Alice", "Bob" };
    public async IAsyncEnumerable<string> GetDataAsync()
    {
        for (int i = 0; i < data.Length; i++)
        {
            Console.WriteLine($"Получаем {i + 1} элемент");
            await Task.Delay(500);
            yield return data[i];
        }
    }
}
```

Для упрощения примера данные здесь представлены в виде простого внутреннего массива строк. Для имитации задержки в получении применяется метод Task.Delay.

Получим эти данные в программе:

```
class Program
{
    static async Task Main(string[] args)
    {
        Repository repo = new Repository();
        IAsyncEnumerable<string> data = repo.GetDataAsync();
        await foreach (var name in data)
        {
            Console.WriteLine(name);
        }
    }
}
```

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный

(x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.

2. Цели лабораторной работы.

3. Ответы на контрольные вопросы.

4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Асинхронный стрим.

2. Задержка.

7. Задание

Добавить пример асинхронного стрима в своё приложение.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №12. Введение в Parallel LINQ

1. Цель и содержание

Цель лаботаторной работы: Рассмотреть Parallel LINQ и метод AsParallel.

2. Теоретическая часть

По умолчанию все элементы коллекции в LINQ обрабатываются последовательно, но начиная с .NET 4.0 в пространство имен System.Linq был добавлен класс ParallelEnumerable, который инкапсулирует функциональность PLINQ (Parallel LINQ) и позволяет выполнять обращения к коллекции в параллельном режиме.

При обработке коллекции PLINQ использует возможности всех процессоров в системе. Источник данных разделяется на сегменты, и каждый сегмент обрабатывается в отдельном потоке. Это позволяет произвести запрос на многоядерных машинах намного быстрее.

В то же время по умолчанию PLINQ выбирает последовательную обработку данных. Переход к параллельной обработке осуществляется в том случае, если это приведет к ускорению работы. Однако, как правило, при параллельных операциях возрастают дополнительные издержки. Поэтому если параллельная обработка потенциально требует больших затрат ресурсов, то PLINK в этом случае может выбрать последовательную обработку, если она не требует больших затрат ресурсов.

Поэтому смысл применения PLINQ имеется преимущественно на больших коллекциях или при сложных операциях, где действительно выгода от распараллеливания запросов может перекрыть возникающие при этом издержки.

Также следует учитывать, что при доступе к общему разделяемому состоянию в параллельных операциях будет неявно использоваться синхронизация, чтобы избежать взаимоблокировки доступа к этим общим ресурсам. Затраты на синхронизацию ведут к снижению производительности,

поэтому желательно избегать или ограничивать применения в параллельных операциях разделяемых ресурсов.

Метод AsParallel() позволяет распараллелить запрос к источнику данных. Он реализован как метод расширения LINQ у массивов и коллекций. При вызове данного метода источник данных разделяется на части (если это возможно) и над каждой частью отдельно производятся операции.

Рассмотрим простейший пример нахождения факториалов чисел:

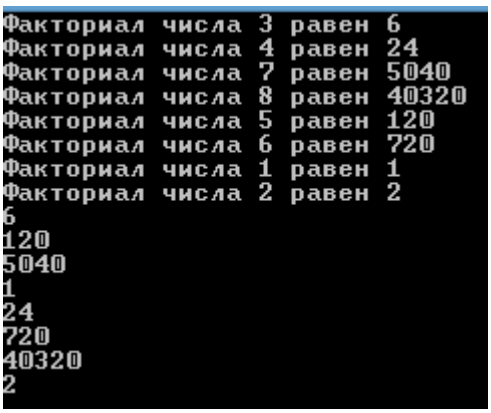
```
static void Main(string[] args)
{
    int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, };
    var factorials = from n in numbers.AsParallel()
                             select Factorial(n);
    foreach (var n in factorials)
        Console.WriteLine(n);
    Console.ReadLine();
}

static int Factorial(int x)
{
    int result = 1;

    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    Console.WriteLine($"Факториал числа {x} равен {result}");
    return result;
}
```

Фактически здесь обычный запрос LINQ, только к источнику данных применяется метод AsParallel.

Результат работы программы говорит о том, что данные выбирались для нахождения факториала не последовательно. То есть произошло распараллеливание работы программы:



Аналогичная операция с помощью методов расширения:

```
var factorials = numbers.AsParallel().Select(x => Factorial(x));
```

Выше рассмотренный код по подсчету факториала можно еще больше оптимизировать с точки зрения параллелизации. В частности, для вывода результата параллельной операции используется цикл foreach. Но его использование приводит к увеличению издержек - необходимо склеить полученные в разных потоках данные в один набор и затем их перебрать в цикле. Более оптимально в данном случае было бы использование метода ForAll(), который выводит данные в том же потоке, в котором они обрабатываются:

```
int[] numbers = new int[] { -2, -1, 0, 1, 2, 4, 3, 5, 6, 7, 8, };
(from n in numbers.AsParallel()
 where n >0
 select Factorial(n))
 .ForAll(Console.WriteLine);
```

Метод ForAll() в качестве параметра принимает делегат Action, который указывает на выполняемое действие.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение:

операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.
2. Цели лабораторной работы.
3. Ответы на контрольные вопросы.
4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Метод AsParallel.
2. Метод ForAll.

7. Задание

Рассмотренный ParallelLINQ использовать во всех своих методах.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №13. ParallelLINQ. Метод AsOrdered

1. Цель и содержание

Цель лабототаторной работы: Рассмотреть метод AsOrdered.

2. Теоретическая часть

При выполнении параллельного запроса порядок данных в результирующей выборки может быть не предсказуем. Например:

```
static void Main(string[] args)
{
    int[] numbers = new int[] { -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, };
    var factorials = from n in numbers.AsParallel()
                     where n >0
                     select Factorial(n);
    foreach (var n in factorials)
        Console.WriteLine(n);

    Console.Read();
}
static int Factorial(int x)
{
    int result = 1;

    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    return result;
}
```

Программа может дать нам следующий результат:

```
5040
1
40320
2
24
6
120
720
```

То есть данные склеиваются в общий набор неупорядоченно.

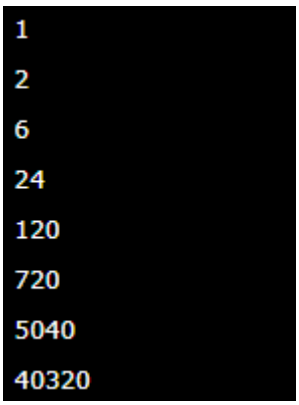
Если в запросе применяются операторы или методы сортировки в запросе, данные автоматически упорядочиваются:

```
var factorials = from n in numbers.AsParallel()
                 where n >0
                 orderby n
                 select Factorial(n);
```

Однако не всегда оператор orderby или метод OrderBy используются в запросах. Более того они упорядочивают результирующую выборку в соответствии с результатами, а не в соответствии с исходной последовательностью. В этих случаях мы может применять метод AsOrdered():

```
var factorials = from n in numbers.AsParallel().AsOrdered()
                 where n >0
                 select Factorial(n);
```

В этом случае результат уже будет упорядоченным в соответствии с тем, как элементы располагаются в исходной последовательности:



В то же время надо понимать, что упорядочивание в параллельной операции приводит к увеличению издержек, поэтому подобный запрос будет выполняться медленнее, чем неупорядоченный. И если задача не требует возвращение упорядоченного набора, то лучше не применять метод AsOrdered.

Кроме того, если в программе предстоят какие-нибудь манипуляции с полученным набором, однако упорядочивание больше не требуется, мы можем применить метод AsUnordered():

```
var factorials = from n in numbers.AsParallel().AsOrdered()
                 where n >0
                 select Factorial(n);
var query = from n in factorials.AsUnordered()
            where n > 100
            select n;
```

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.

2. Цели лабораторной работы.

4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Задание

Один из методов, использованныъ в Вашем приложении использовать с методом AsOrdered.

7. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) поясняет ход выполнения индивидуального задания;

2) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.

Лабораторная работа №14. Обработка ошибок и отмена операции

1. Цель и содержание

Цель лабототаторной работы: Рассмотреть обработку ошибок.

2. Теоретическая часть

При выполнении параллельных операций также могут возникать ошибки, обработка которых имеет свои особенности. При параллельной обработке коллекция разделяется а части, и каждая часть обрабатывается в отдельном потоке. Однако если возникнет ошибка в одном из потоков, то система прерывает выполнение всех потоков.

При генерации исключений все они агрегируются в одном исключении типа AggregateException

Например, пусть в метод факториала передается массив объектов, который содержит не только числа, но и строки:

```
object[] numbers2 = new object[] { 1, 2, 3, 4, 5, "hello" };

var factorials = from n in numbers2.AsParallel()
                 let x = (int)n
                 select Factorial(x);
try
{
    factorials.ForAll(n => Console.WriteLine(n));
}
catch (AggregateException ex)
{
    foreach (var e in ex.InnerExceptions)
    {
        Console.WriteLine(e.Message);
    }
}
```

Так как массив содержит строку, то попытка приведения закончится неудачей, и при запуске приложения в Visual Studio в режиме отладки выполнение остановится на строке преобразования. А после продолжения сработает перехват исключения в блоке catch, и на консоль будет выведено сообщение об ошибке.

Прерывание параллельной операции

Вполне вероятно, что нам может потребоваться прекратить операцию до ее завершения. В этом случае мы можем использовать метод WithCancellation(), которому в качестве параметра передается токен CancellationToken:

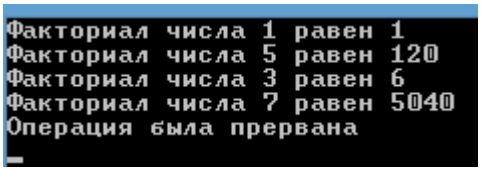
```
static void Main(string[] args)
{
    CancellationTokenSource cts = new CancellationTokenSource();
    new Task(() =>
    {
        Thread.Sleep(400);
        cts.Cancel();
    }).Start();
    try
    {
        int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, };
        var factorials = from n in numbers.AsParallel().WithCancellation(cts.Token)
                         select Factorial(n);
        foreach (var n in factorials)
            Console.WriteLine(n);
    }
    catch(OperationCanceledException ex)
    {
        Console.WriteLine("Операция была прервана");
    }
    catch (AggregateException ex)
    {
        if (ex.InnerExceptions != null)
        {
            foreach (Exception e in ex.InnerExceptions)
                Console.WriteLine(e.Message);
        }
    }
    finally
    {
        cts.Dispose();
    }
    Console.ReadLine();
}

static int Factorial(int x)
{
    int result = 1;

    for (int i = 1; i <= x; i++)
    {
        result *= i;
    }
    Console.WriteLine($"Факториал числа {x} равен {result}");
    Thread.Sleep(1000);
    return result;
}
```

В параллельной запущенной задаче вызывается метод cts.Cancel(), что приводит к завершению операции и генерации исключения OperationCanceledException:

Отмена задачи с помощью метода WithCancellation:



При этом также имеет смысл обрабатывать исключение AggregateException, так как если параллельно возникает еще одно исключение, то это исключение, а также OperationCanceledException помещаются внутрь одного объекта AggregateException.

3. Аппаратура и материалы

Для выполнения лабораторной работы рекомендуется использовать персональный компьютер со следующими характеристиками: 32-разрядный (x86) или 64-разрядный (x64) процессор с тактовой частотой 1,5 ГГц и выше, оперативная память – 1 Гб и выше, свободное дисковое пространство – не менее 1 Гб, графическое устройство DirectX 9. Программное обеспечение: операционная система на базе WINDOWS, Microsoft Visual Studio 2008 и старше.

4. Указания по технике безопасности

Техника безопасности при выполнении лабораторной работы совпадает с общепринятой для пользователей персональных компьютеров. Самостоятельно не производить ремонт персонального компьютера, установку и удаление программного обеспечения; в случае неисправности персонального компьютера сообщить об этом обслуживающему персоналу лаборатории (оператору, администратору); соблюдать правила техники безопасности при работе с электрооборудованием; не касаться электрических розеток металлическими предметами; рабочее место пользователя персонального компьютера должно содержаться в чистоте; не разрешается возле персонального компьютера принимать пищу, напитки.

5. Содержание отчета и его форма

1. Номер и название лабораторной работы.

2. Цели лабораторной работы.

3. Ответы на контрольные вопросы.

4. Экранные формы, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения:

Отчет о выполнении лабораторной работы в письменном виде сдается преподавателю.

6. Вопросы для защиты работы

1. Прерывание параллельной операции.

2. Обработка ошибок.

7. Задание

Создать централизованную обработку ошибок во всех методах Вашего приложения.

8. Защита работы

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание.

Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент:

1) отвечает на контрольные вопросы;

2) поясняет ход выполнения индивидуального задания;

3) поясняет результаты, полученные в результате выполнения индивидуального задания.

Ход защиты лабораторной работы контролируется преподавателем.