Ильяхова Алиса, Б9122-02.03.01сцт

Лабораторная работа №2. Применение асинхронных делегатов для реализации многопоточности

Вариант 9

1. Цель и задачи работы

Цель:

Научиться использовать делегаты для организации многопоточного приложения.

Задачи:

- Научиться объявлять, инициализировать и запускать потоки с использованием пользовательских делегатов;
- Научиться запускать потоки с использованием библиотечных делегатов Action<T> и Func<T>;
- Научиться запускать параллельные потоки с использованием лямбда-выражений.

2. Реализация индивидуального задания

2.1. Условие варианта 9

Согласно таблице индивидуальных заданий (стр. 19 методических указаний):

- **Тип делегата**: *лямбда-выражение*
- **Решаемая задача**: метод возвращает результат шифрования строки: каждый исходный символ строки заменяется шифрованным символом, код которого на n больше кода исходного символа.
- **Входные параметры**: два параметра — исходная строка (string) и целое число сдвига (int).

2.2. Объявление делегата и метода шифрования

Для соответствия условию был объявлен пользовательский делегат:

```
delegate string EncryptDelegate(string input, int shift);
```

Реализован метод EncryptString, выполняющий побайтовое шифрование строки:

```
static string EncryptString(string input, int shift)
    {
        if (input == null) return null;
        char[] buffer = new char[input.Length];
        for (int i = 0; i < input.Length; i++)
        {
            buffer[i] = (char)(input[i] + shift);
        }
        return new string(buffer);
    }
```

Метод корректно обрабатывает все символы строки, включая знаки препинания и пробелы, путём сдвига их ASCII-кодов.

2.3. Асинхронный вызов через Task.Run

Для реализации асинхронности использован современный подход на основе класса Task:

```
Task<string> task = Task.Run(() => encryptDel(originalText, shiftValue));
```

Этот вызов:

- Запускает выполнение делегата в фоновом потоке из пула потоков;
- Не блокирует основной поток;
- Возвращает объект Task<string>, позволяющий отслеживать состояние выполнения и получать результат.

2.4. Мониторинг выполнения (опрос состояния)

В основном потоке реализован цикл опроса с использованием свойства IsCompleted:

```
while (!task.IsCompleted)
        {
            Console.Write($"\rОжидание... ({++counter} сек)");
            Thread.Sleep(1000);
        }
```

Это демонстрирует параллельное выполнение: основной поток продолжает работу, в то время как шифрование происходит в фоне.

2.5. Получение результата

После завершения фоновой операции результат извлекается через свойство Result:

```
string encryptedText = task.Result;
```

2.6. Демонстрация через async/await

Дополнительно реализован альтернативный стиль асинхронного программирования:

```
static async Task AsyncDemo(string text, int shift)
    {
        EncryptDelegate del = EncryptString;
        string result = await Task.Run(() => del(text, shift));
        Console.WriteLine($"[async/await] Результат: \"{result}\"");
    }
```

Этот подход обеспечивает неблокирующее ожидание завершения операции и является современным стандартом в .NET.

3. Ответы на контрольные вопросы

1. **Поясните назначение типа IAsyncResult.**

IAsyncResult — интерфейс, представляющий состояние асинхронной операции в устаревшей модели асинхронного программирования (APM). Он позволяет проверять завершение операции (IsCompleted), дожидаться её окончания (AsyncWaitHandle.WaitOne()) и передавать дополнительные данные (AsyncState). В современных приложениях вместо него используются Task и async/await.

2. **Для чего используется метод Thread.Sleep()?**

Thread.Sleep() приостанавливает выполнение текущего потока на указанное время (в миллисекундах). В данной программе он используется:

   a. Для имитации длительной работы (в реальных задачах — ожидание ввода, сетевых операций и т.п.);
   b. Для демонстрации параллелизма между основным и фоновым потоками.

3. **Поясните механизм возврата значения из метода асинхронного делегата.**

В устаревшей модели значение возвращается через метод EndInvoke(). В современной реализации (через Task<T>) результат доступен через свойство Result или оператор await. Оба подхода обеспечивают синхронизацию: если операция ещё не завершена, вызывающий поток блокируется до её окончания.

4. **Как произвести возврат более одного значения из метода?**

Возможные способы:

a. Использовать кортеж ((T1, T2)), доступный в C# 7.0+;
b. Вернуть объект пользовательского класса или структуры;
c. Использовать параметры out или ref (но они несовместимы с Task.Run и делегатами Func<T>).

5. **Какая разница существует между библиотечными делегатами, пользовательскими типами делегатов и лямбда-выражениями? Являются ли эти делегаты взаимозаменяемыми при реализации асинхронного вызова методов?**
   a. **Пользовательские делегаты** — явно объявленные (delegate ...), повышают читаемость кода.
   b. **Библиотечные делегаты** (Func<T>, Action<T>) — универсальные, сокращают объём кода.
   c. **Лямбда-выражения** — синтаксический сахар для создания анонимных методов.

Все три подхода взаимозаменяемы, если сигнатуры совпадают. В данной работе использован пользовательский делегат для соответствия формулировке задания, но его можно заменить на Func<string, int, string> без изменения логики.

4. Экранные формы и листинг программы

4.1. Консольный вывод программы

```
=== Лабораторная работа №2. Вариант 9 ===
Шифрование строки с использованием асинхронного делегата (через Task)

Исходная строка: "Hello, World!"
Сдвиг: 3

Основной поток: запуск шифрования в фоновом потоке...
Ожидание... (1 сек)

Зашифрованная строка: "Khoor/#Zruog$"

=== Демонстрация через async/await ===
[async/await] Результат: "Khoor/#Zruog$"

=== Готово ===
```

Обратите внимание: символы ,, пробел и ! также сдвинулись (например, ! -> $), что подтверждает корректность побайтового шифрования.

4.2. Полный листинг программы с комментариями

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

// Лабораторная работа №2. Вариант 9
// Современная реализация асинхронности через Task (вместо BeginInvoke)

class Program
{
    // Метод шифрования строки
    static string EncryptString(string input, int shift)
    {
        if (input == null) return null;
        char[] buffer = new char[input.Length];
        for (int i = 0; i < input.Length; i++)
        {
            buffer[i] = (char)(input[i] + shift);
        }
        return new string(buffer);
    }

    // Делегат, соответствующий методу
    delegate string EncryptDelegate(string input, int shift);

    static void Main(string[] args)
    {
        Console.WriteLine("=== Лабораторная работа №2. Вариант 9 ===");
        Console.WriteLine("Шифрование строки с использованием асинхронного делегата (через
Task)\n");

        string originalText = "Hello, World!";
        int shiftValue = 3;

        Console.WriteLine($"Исходная строка: \"{originalText}\"");
        Console.WriteLine($"Сдвиг: {shiftValue}\n");

        // Создаём экземпляр делегата
        EncryptDelegate encryptDel = EncryptString;

        // === Асинхронный вызов через Task.Run ===
        Console.WriteLine("Основной поток: запуск шифрования в фоновом потоке...");
        Task<string> task = Task.Run(() => encryptDel(originalText, shiftValue));

        // Имитация работы основного потока
        int counter = 0;
        while (!task.IsCompleted)
        {
            Console.Write($"\rОжидание... ({++counter} сек)");
            Thread.Sleep(1000);
```

```
        }

        // Получаем результат
        string encryptedText = task.Result;
        Console.WriteLine($"\n\nЗашифрованная строка: \"{encryptedText}\"");

        // === Демонстрация через async/await (альтернативный стиль) ===
        Console.WriteLine("\n=== Демонстрация через async/await ===");
        AsyncDemo(originalText, shiftValue).Wait(); // дожидаемся завершения

        Console.WriteLine("\n=== Готово ===");
    }

    static async Task AsyncDemo(string text, int shift)
    {
        EncryptDelegate del = EncryptString;
        string result = await Task.Run(() => del(text, shift));
        Console.WriteLine($"[async/await] Результат: \"{result}\"");
    }
}
```

5. Вывод

В ходе выполнения лабораторной работы были:

- Реализован метод шифрования строки с заданным сдвигом;
- Продемонстрирована многопоточность через современные средства .NET (Task, async/await);
- Показаны два стиля асинхронного программирования: опрос состояния (IsCompleted) и неблокирующее ожидание (await);
- Подтверждена возможность использования делегатов для гибкой передачи поведения в фоновые задачи.