

ГУАП

КАФЕДРА № 44

ОТЧЕТ
ЗАЩИЩЕН С
ОЦЕНКОЙ
ПРЕПОДАВАТЕЛЬ

доц., канд. техн. наук, доц.

должность, уч. степень, звание

подпись, дата

Н.В. Кучин

инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №2

ПРОГРАММИРОВАНИЕ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ

по курсу: ОПЕРАЦИОННЫЕ СИСТЕМЫ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №

4143

подпись, дата

Е.Д.Тегай

инициалы, фамилия

Санкт-Петербург 2023

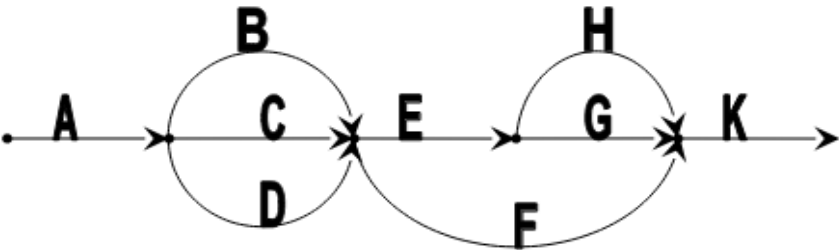
Цель работы

Необходимо написать и отладить программу, которая реализует параллельное выполнение нескольких задач, каждая из которых решает некоторую заданную функцию.

Индивидуальное задание

Содержание индивидуального задания под №1 продемонстрировано на рисунке 1.

ЗАДАНИЕ № 1



Имя задачи	Длительность	Приоритет	Функция
A	1	0	Генерирует $M[1..n, 1..n]$, $R[1..n]$ - integer
B	1	1	$f_1(M, R)$
C	1	1	$f_2(M, R)$
D	1	1	$f_3(M, R)$
E	1	2	$f_4(f_1, f_2, f_3)$
F	2	3	$f_5(f_1, f_2, f_3)$
G	1	3	$f_6(f_4)$
H	1	3	$f_7(f_4)$
K	1	4	$f_8(f_5, f_6, f_7)$

Рисунок 1 — Индивидуальное задание

Используемые методы

Для реализации поставленной задачи в качестве языка программирования был выбран C#, так как он довольно прост в использовании, предоставляет множество инструментов для обеспечения безопасности потоков (lock-конструкции, мониторы, семафоры, мьютексы), что позволяет избежать состояния «гонки» и других проблем многопоточности.

Синхронизация параллельных процессов (потоков) была программно реализована с помощью такого механизма, как семафор. Он предназначен для управления доступом к ресурсам в многозадачном окружении. Семафор предоставляет возможность ограничивать количество потоков, которые могут одновременно получить доступ к определённому ресурсу или выполнять критическую секцию кода (та часть кода, где происходит доступ к общему ресурсу, который может быть использован несколькими потоками). В отличие от мьютекса, к объекту могут обращаться несколько потоков, а не один. Ещё одно отличие заключается в том, что мьютекс может разблокировать только тот поток, который его заблокировал, а семафор может разблокировать любой поток.

Графически семафор изображён на рисунке 2. В данном случае семафор имеет такое ограничение: к объекту одновременно могут подключиться только 2 потока (они обозначены зелёным, ожидающий поток обозначен красным).

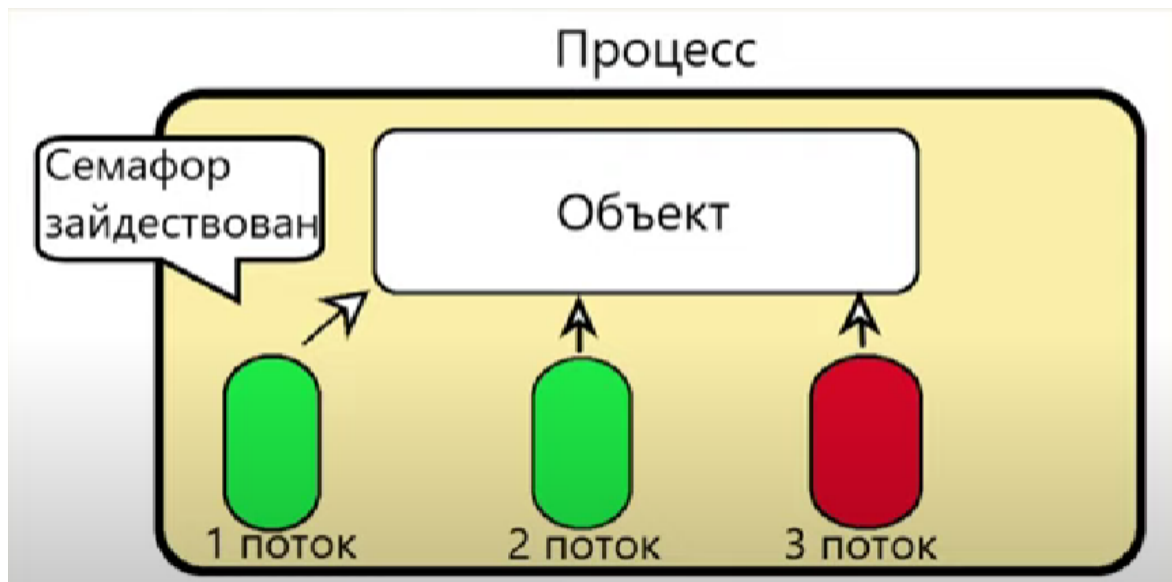


Рисунок 2 - Семафор

Перейдём непосредственно к рассмотрению используемых методов.

Интерфейс

Лабораторная работа выполнялась в такой интегрированной среде разработки, как Visual Studio, где можно создать приложение Windows Forms. WinForms – библиотека для разработки графических пользовательских интерфейсов (GUI). С помощью неё можно создавать оконные приложения с использованием различных элементов управления (кнопки, текстовые поля, таблицы, изображения и другие). Созданный интерфейс изображён на рисунке 3.

Form1

ЗАДАЧИ

Запустить задачу

Очистить поле

A
B
C
D
E
F
G
H
K

Large white text area with a scrollbar.

Рисунок 3 — Созданный интерфейс

Рассмотрим его подробнее. Синие окошки с именами «Запустить задачу» и «Очистить поле» - кнопки, которые позволяют запустить первую задачу согласно индивидуальному варианту и очистить текстовое поле соответственно. Они создавались с помощью панели элементов, вид которой изображён на рисунке 4.

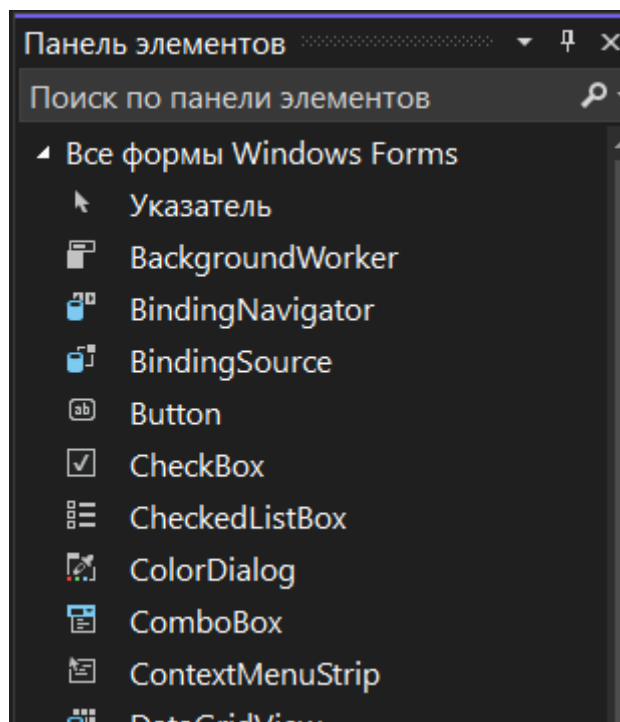


Рисунок 4 — Панель элементов

Собственно, все составляющие окна приложения были добавлены с помощью панели элементов. Шрифт, цвет поля, размер текста и иные внешние характеристики устанавливались с помощью окна со всеми параметрами объекта. Пример такого окна изображён на рисунке 5 (параметры кнопки «Запустить задачу»).

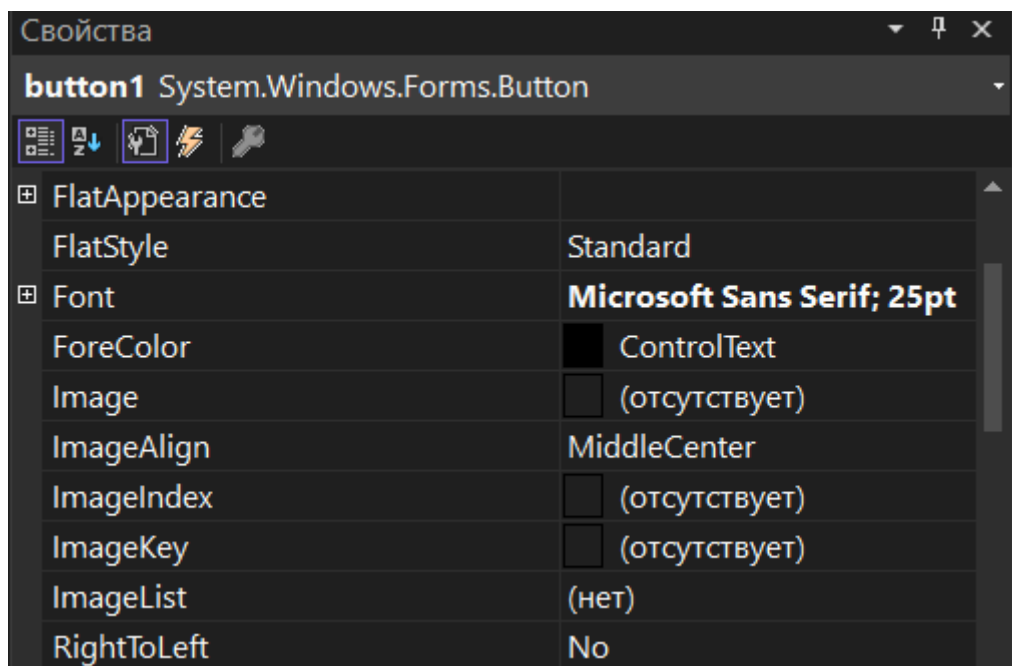
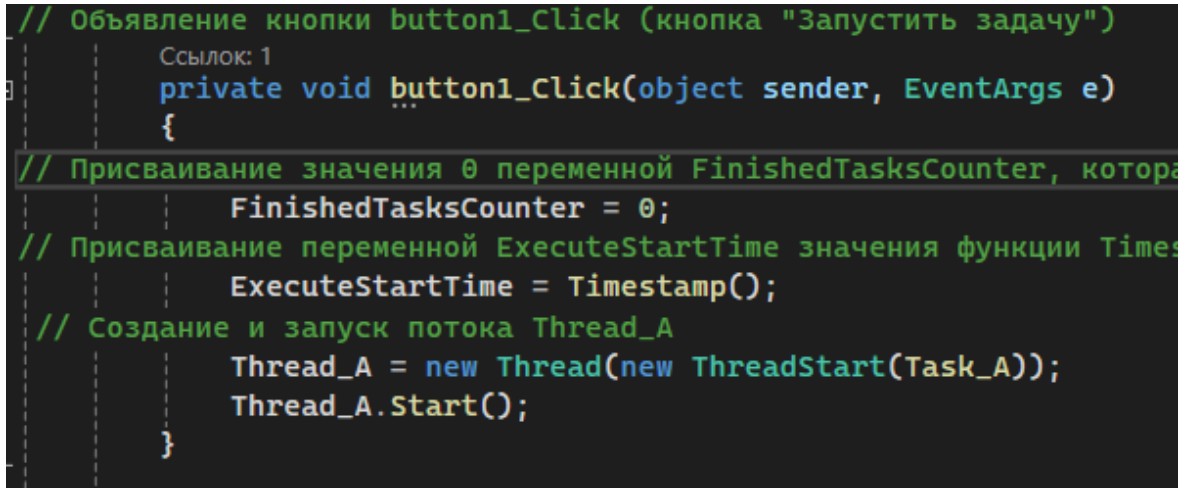


Рисунок 5 — Свойства кнопки «Запустить задачу»

Как только в окно приложения в режиме конструктора добавляется какой-либо объект, в коде также добавляется соответствующий метод. На рисунке 6 изображён пример секции кода, относящемуся к кнопке «Запустить задачу».

A screenshot of a code editor showing a C# method named button1_Click. The code is written in a dark-themed editor with syntax highlighting. The method is private and takes two parameters: object sender and EventArgs e. It contains several lines of code: a comment about assigning 0 to FinishedTasksCounter, a line assigning 0 to FinishedTasksCounter, a comment about assigning ExecuteStartTime, a line assigning ExecuteStartTime to Timestamp(), a comment about creating and starting Thread_A, a line creating Thread_A with new Thread and new ThreadStart(Task_A), and a line starting Thread_A. The code is enclosed in curly braces.

```
// Объявление кнопки button1_Click (кнопка "Запустить задачу")
// Ссылка: 1
private void button1_Click(object sender, EventArgs e)
{
    // Присваивание значения 0 переменной FinishedTasksCounter, которая
    FinishedTasksCounter = 0;
    // Присваивание переменной ExecuteStartTime значения функции Times
    ExecuteStartTime = Timestamp();
    // Создание и запуск потока Thread_A
    Thread_A = new Thread(new ThreadStart(Task_A));
    Thread_A.Start();
}
```

Рисунок 6 — Метод, относящийся к кнопке «Запустить задачу»

На самом деле внутри объявленного метода изначально ничего не было. Весь код внутри был прописан для придания функциональности кнопке. Когда добавляется какой-либо элемент, всегда создаётся соответствующий метод. Его сопровождают 2 аргумента — object sender и EventArgs e. Это также можно рассмотреть и на рисунке 7. Первый аргумент обозначает объект, инициировавший событие (в данном случае — сама кнопка), а второй предоставляет информации о событии (например, тип события).

Таким же образом были добавлены:

- сама форма, на которой располагались все объекты;
- кнопка «Очистить поле»;
- прогресс-бары для каждой задачи;
- названия для прогресс-баров — имена задач (для более приятного восприятия и доступности понимания);

- текстовое поле, куда записывалась информация о начале работы задачи, её завершении, а также о промежуточных вычислениях функций.

В добавление: функционал также был добавлен и кнопке «Очистить поле». Секция кода, относящаяся к этой кнопке изображена на рисунке 7.

```
private void button2_Click(object sender, EventArgs e)
{
    // Очищение текстового поля
    textBox1.Text = "";
    // Обнуление всех ProgressBar
    progressBar1.Value = 0;
    progressBar2.Value = 0;
    progressBar3.Value = 0;
    progressBar4.Value = 0;
    progressBar5.Value = 0;
    progressBar6.Value = 0;
    progressBar7.Value = 0;
    progressBar8.Value = 0;
    progressBar9.Value = 0;
}
```

Рисунок 7 — Метод, относящийся к кнопке «Очистить поле»

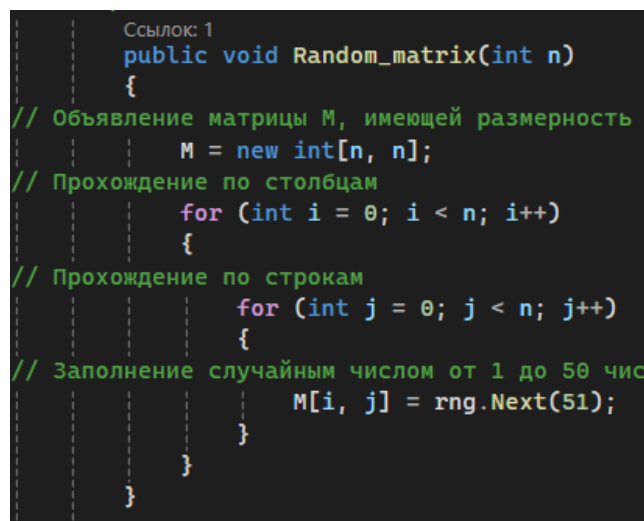
Внутри этого метода содержатся такие действия, как очищение поля (превращение содержимого поля в пустую строку, каким бы оно ни было до нажатия на эту кнопку) и обнуление всех прогресс-баров (то бишь они станут «заполнены» на 0%).

Всем остальным объектам не было присвоено никакого функционала, так что соответствующие им методы пусты в своём внутреннем содержании.

Методы

Для реализации поставленной задачи были использованы такие методы, как:

1) `Random_matrix(int n)`. Его содержимое продемонстрировано на

A screenshot of a code editor showing the implementation of the `Random_matrix` method in C#. The code is as follows:

```
Ссылка: 1
public void Random_matrix(int n)
{
    // Объявление матрицы М, имеющей размерность
    М = new int[n, n];
    // Прохождение по столбцам
    for (int i = 0; i < n; i++)
    {
        // Прохождение по строкам
        for (int j = 0; j < n; j++)
        {
            // Заполнение случайным числом от 1 до 50 чис
            М[i, j] = rng.Next(51);
        }
    }
}
```

рисунке 8.

Рисунок 8 — Метод `Random_matrix(int n)`

Данный метод генерирует матрицу размером $n \times n$ с помощью генератора псевдослучайных чисел. В качестве параметра передаётся лишь заранее определённая переменная n (её значение равно 4), которая в дальнейшем используется для задания размерности будущей матрицы M . Она также является граничным значением в двух циклах `for`, с помощью которых матрица M и заполняется полностью (два цикла обеспечивают прохождение как по всем строкам, так и по всем столбцам) псевдослучайными числами. Следует отметить, что псевдослучайные числа брались из диапазона 1 — 50 включительно.

2) `R_generation(int n)`. Его содержимое продемонстрировано на рисунке 9.

```

Ссылка: 1
public void R_generation(int n)
{
    //объявление переменной R – массива с размер
    R = new int[n];
    //прохождение по столбцам
    for (int i = 0; i < n; i++)
    {
        //присваивание каждому элементу массива сл
        R[i] = rng.Next(51);
    }
}

```

Рисунок 9 — Метод Random_matrix(int n)

Данный метод схож по своей задаче с прошлым методом — он заполняет массив R псевдослучайными числами. В качестве параметра также передаётся ранее определённая переменная n, равная 4, с помощью которой задаётся размерность будущего массива R и которая так же является граничным значением внутри цикла for (уже одиночного, так как массив R — одномерный), который и обеспечивает заполнение всего массива псевдослучайными числами, стоящими в диапазоне 1 — 50 включительно.

3) UpdateText(String what). Его содержимое продемонстрировано на рисунке 10.

```

public void UpdateText(String what)
{
    // вызов произошел из потока, инвокируем в главном потоке.
    // щем, это проверка на то, выполняется ли текущий код в потоке, в котором был с
    // ент управления – Form1. Invoke используется для безопасного доступа к элемент
    if (InvokeRequired)
    {
        // гарантирует, что LogText будет выполнен в потоке, в котором был создан эле
        this.Invoke(new Action<string>(UpdateText), new object[] { what });
        return;
    }
    // текстовое поле пустое, то перенос строки в начало не производится
    String pre = "\r\n";
    if (textBox1.Text == "")
    {
        pre = "";
    }
    // вление текстового поля с учетом вызова из разных потоков
    textBox1.Text += pre + what.ToString();
}

```

Рисунок 10 - Метод UpdateText(String what)

Данный метод позволяет обновлять текстовое поле (объект `textBox1` в конструкторе). Проще говоря, с помощью этого метода выводится текст с информацией о начале или завершении какой-либо задачи или же о результатах вычислений.

В качестве параметра передаётся строка `what`. Первое условие проверяет, вызван ли код из потока, отличного от потока пользовательского интерфейса. Если нет, то выполняется метод `Invoke`. Он используется для синхронизации выполнения кода с потоком пользовательского интерфейса. Собственно, он принимает делегат `Action<string>` (метод без возвращаемого значения), представляющий метод `UpdateText` (делегат — такой тип данных, который предоставляет ссылку на метод).

Затем создаётся строка `pre`, которая состоит из комбинации символов `\r\n` (символ возврата каретки в начало строки и перенос на новую строку). После этого идёт вторая проверка — проверка на пустое текстовое поле. Если поле пустое, то переменная `pre` превращается в

просто пустую строку. После этого значение переменной `what` преобразуется в строку, предварительно добавляя значение переменной `pre` в зависимости от исхода проверки.

4) `UpdateProgressBar(ProgressBar progressBar, int value)`. Его содержимое продемонстрировано на рисунке 11.

```
private void UpdateProgressBar(ProgressBar progressBar, int value)
{
    if (progressBar.InvokeRequired)
    {
        // той же логике, как и в методе LogText, если вызов происходит из потока, отличного от главного,
        // для безопасности используется Invoke */
        progressBar.Invoke(new Action<ProgressBar, int>(UpdateProgressBar), progressBar, value);
        return;
    }
    // присвоение нового значения для ProgressBar в соответствии с переданным параметром value
    progressBar.Value = value;
}
```

Рисунок 11 — Метод `UpdateProgressBar(ProgressBar progressBar, int value)`

Данный метод обновляет значение прогресс-баров в графическом интерфейсе. Логика метода построена по аналогии с предыдущим методом: если вызов происходит из потока, отличного от главного, в целях безопасности и синхронизации используется `Invoke`. Затем устанавливается значение прогресс-бара в соответствии с переданным параметром `value`.

В качестве параметров передаётся объект типа `ProgressBar`, представляющий собой элемент управления полосой прогресса. Также передаётся и `in value` – численное значение прогресса, которое будет установлено на прогресс-баре.

5) `TimeStamp()`. Его содержимое продемонстрировано на рисунке 12.

```

public long Timestamp()
{
    // Возвращение текущей метки времени (в мс)
    return DateTimeOffset.UtcNow.ToUnixTimeMilliseconds();
}

```

Рисунок 12 — Метод TimeStamp()

Данный метод возвращает текущее время в мс с начала эпохи Unix (1 января 1970 года) в формате всемирного координированного времени.

6) Information_of_Thread(String ActiveThread, String ParentThread). Его содержимое продемонстрировано на рисунке 13.

```

public long Information_of_Thread(String ActiveThread, String ParentThread)
{
    // Запись информации о запуске задачи (какая задача запущена, с помощью какой задачи она была запущена и в какой момент времени (в мс))
    UpdateText(" Задача " + ActiveThread + " запущена с помощью задачи " + ParentThread + " в " + (Timestamp() - ExecuteStartTime).ToString() + " мс ");
    // Возвращение значения метода Timestamp (в мс)
    return Timestamp();
}

```

Рисунок 13 — Метод Information_of_Thread(String ActiveThread, String ParentThread)

Данный метод используется для вывода информации о запущенном потоке. В качестве параметров передаётся String ActiveThread – имя текущего потока и String ParentThread – имя родительского потока, то есть того потока, благодаря которому и был запущен текущий.

Внутри метода вызывается ранее рассмотренный метод UpdateText, с помощью которого выводится вся необходимая информация о запущенном потоке. Затем идёт возвращение так же ранее рассмотренного метода TimeStamp, с помощью которого выводится текущее время в мс с начала эпохи Unix.

7) Information_Of_Finished_Task(String ActiveThread, String StartTime). Его содержимое продемонстрировано на рисунке 14.

```

public void Information_Of_Finished_Task(String ActiveThread, long StartTime)
{
    // вызов происходит из потока, отличного от главного, в целях безопасности используется Invoke
    if (InvokeRequired)
    {
        this.Invoke(new Action<string, long>(Information_Of_Finished_Task), new object[]
        { ActiveThread, StartTime });
        return;
    }
    // инициализация переменной NowTime значения текущего времени (в мс)
    long NowTime = Timestamp();
    // время выполнения задачи
    String ExecTime = (NowTime - StartTime).ToString();
    // Запись информации о завершении задачи
    UpdateText(" Задача " + ActiveThread + " завершилась.\r\nВремя выполнения: " + ExecTime + " мс.\r\nВремя окончания: " + (NowTime - ExecuteStartTime).ToString() + " мс.\r\n");
}

```

Рисунок 14 — Метод Information_Of_Finished_Task(String ActiveThread, String StartTime)

Данный метод позволяет выводить информацию о завершённой задаче. В качестве параметров передаётся String ActiveThread — имя текущего потока и String StartTime — строковое значение времени начала выполнения задачи в мс.

Внутри метода используется всё та же проверка — вызван ли код из потока, отличного от потока пользовательского интерфейса. Если нет, то используется метод Invoke.

Затем создаётся переменная NowTime, которой присваивается значение ранее рассмотренного метода Timestamp. Создаётся и строка ExecTime, которой присваивается строковое значение разности переменной NowTime и StartTime — времени выполнения задачи. После этого вызывается ранее рассмотренный метод UpdateText, с помощью которого выводится вся необходимая информация о завершённой задаче.

Функции

Также для реализации поставленной задачи были созданы методы, соответствующие задачам из индивидуального варианта. Рассмотрим их подробнее.

1) F1 (int[,] M, int[] R). Её содержимое продемонстрировано на рисунке 15.

```

public int F1(int[,] M, int[] R)
{
    // инициализация переменной summa, которая будет хранить в себе значение
    int summa = 0;
    // значение 1-го элемента массива R - element
    int element = R[1];
    // инициализация переменной Intermediate_Summa - промежуточная сумма.
    // переменная будет хранить в себе значение некой математической
    int Intermediate_Summa = 0;
    // прохождение по 0-му столбцу массива M (матрицы)
    for (int i = 0; i < 1; i++)
    {
        // прохождение по строкам массива M (матрицы)
        for (int j = 0; j < n; j++)
        {
            // к каждому рассматриваемому элементу матрицы прибавляется значение
            Intermediate_Summa = M[i, j] + element;
            // это значение добавляется (прибавляется) к итоговой сумме
            summa += Intermediate_Summa;
        }
    }
    // возвращение итоговой суммы
    return summa;
}

```

Рисунок 15 — Функция F1 (int[,] M, int[] R)

Данная функция описывает поведение функции F1 из индивидуального варианта. Всё содержимое рассматриваемых в этом блоке функций было придумано разработчиком.

В качестве параметров передаются сгенерированные массивы M и R. Внутри функции создаются такие переменные, как summa, element, Intermediate_Summa, которые отвечают за итоговый вывод суммы, первого элемента массива R и промежуточного значения суммы соответственно. Затем идёт прохождение по нулевому столбцу двумерного массива M, к каждому значению которого прибавляется значение первого элемента одномерного массива R. Полученный на каждой итерации результат сложения присваивается переменной Intermediate_Summa, который потом передаётся в итоговый результат,

хранящийся в переменной `summa`. В результате получаем некое итоговое значение переменной `summa`.

2) F2 (int[,] M, int[] R). Её содержимое продемонстрировано на рисунке 16.

```
public int F2(int[,] M, int[] R)
{
    // инициализация переменной subtraction, которая будет хранить в себе
    // значение 2-го элемента массива R - element
    int subtraction = 100;
    int element = R[2];
    // инициализация переменной Intermediate_subtraction - промежуточная
    // переменная для хранения значения по 1-му столбцу массива M (матрицы)
    int Intermediate_subtraction = 0;
    // прохождение по 1-му столбцу массива M (матрицы)
    for (int i = 1; i < 2; i++)
    {
        // прохождение по строкам массива M (матрицы)
        for (int j = 0; j < n; j++)
        {
            // к каждому рассматриваемому элементу матрицы прибавляется значение
            // element, и это значение вычитается из переменной subtraction
            Intermediate_subtraction = M[i, j] + element;
            subtraction -= Intermediate_subtraction;
        }
    }
    // возвращение переменной subtraction
    return subtraction;
}
```

Рисунок 16 - Функция F2 (int[,] M, int[] R)

Данная функция описывает поведение функции F2 из индивидуального варианта.

В качестве параметров передаются сгенерированные массивы M и R. Внутри функции создаются такие переменные, как `subtraction`, `element`, `Intermediate_Subtraction`, которые отвечают за итоговый вывод разности, второго элемента массива R и промежуточного значения разности соответственно. Затем идёт прохождение по первому столбцу двумерного массива M, к каждому значению которого прибавляется значение второго элемента одномерного массива R. Полученный на

каждой итерации результат сложения присваивается переменной `Intermediate_Substraction`, который потом вычитается из итоговой переменной `substraction`. В результате получаем некое итоговое значение переменной `substraction`.

3) `F3 (int[,] M, int[] R)`. Её содержимое продемонстрировано на рисунке 17.

```
public int F3(int[,] M, int[] R)
{
    // инициализация переменной summa, которая будет хранить в себе
    // значение 3-го элемента массива R - element
    int summa = 0;
    int element = R[3];
    // инициализация переменной Intermediate_Summa - промежуточная
    // переменная будет хранить в себе значение некой матрицы
    int Intermediate_Summa = 0;
    // прохождение по всем диагональным элементам матрицы M
    for (int i = 0; i < 4; i++)
    {
        // для каждого рассматриваемому элементу матрицы прибавляется
        // значение element^2 к Intermediate_Summa
        Intermediate_Summa = M[i, i] + element^2;
        // и это значение добавляется (прибавляется) к итоговой
        // сумме summa
        summa += Intermediate_Summa;
    }
    // возвращение итоговой суммы
    return summa;
}
```

Рисунок 17 — Функция `F3 (int[,] M, int[] R)`

Данная функция описывает поведение функции `F3` из индивидуального варианта.

В качестве параметров передаются сгенерированные массивы `M` и `R`. Внутри функции создаются такие переменные, как `summa`, `element`, `Intermediate_Summa`, которые отвечают за итоговый вывод суммы, третьего элемента массива `R` и промежуточного значения суммы соответственно. Затем идёт прохождение по диагональным значениям двумерного массива `M`, где к каждому прибавляется значение третьего элемента одномерного массива `R` в квадрате. Полученный на каждой

итерации результат сложения присваивается переменной `Intermediate_Summa`, который потом прибавляется к итоговой переменной `summa`. В результате получаем некое итоговое значение переменной `summa`.

4) `F4(int F1, int F2, int F3)`. Её содержимое продемонстрировано на рисунке 18.

```
public int F4(int F1, int F2, int F3)
{
    // инициализация переменной result_F4, которая будет
    // принимать значения функций F1, F2, F3*/
    int result_F4 = F2 + F1 - F3;
    // возвращение итоговой переменной result_F4
    return result_F4;
}
```

Рисунок 18 — Функция `F4(int F1, int F2, int F3)`

Данная функция описывает поведение функции `F4` из индивидуального варианта.

В качестве параметров передаются численные значения функций `F1`, `F2`, `F3`. Внутри функции создаётся результирующая переменная `result_F4`, которой присваивается значение неких математических операций с передаваемыми параметрами.

5) `F5(int F1, int F2, int F3)`. Её содержимое продемонстрировано на рисунке 19.

```
public int F5(int F1, int F2, int F3)
{
    // инициализация переменной result_f5, которая будет
    // принимать значения функций F1, F2, F3*/
    int result_F5 = F1 - F2 + F3;
    // возвращение итоговой переменной result_f5
    return result_F5;
}
```

Рисунок 19 - Функция `F5(int F1, int F2, int F3)`

В качестве параметров передаются численные значения функций F1, F2, F3. Внутри функции создаётся результирующая переменная result_F5, которой присваивается значение неких математических операций с передаваемыми параметрами.

6) F6(int F4). Её содержимое продемонстрировано на рисунке 20.

```
public int F6(int F4)
{
    // создание переменной result_F6, к
    int result_F6 = F4 + 14;
    // возвращение итоговой переменной re
    return result_F6;
}
```

Рисунок 20 - Функция F6(int F4)

В качестве параметра передаётся численное значение функции F4. Внутри функции создаётся результирующая переменная result_F6, которой присваивается значение некой математической операции с передаваемым параметром.

7) F7(int F4). Её содержимое продемонстрировано на рисунке 21.

```
public int F7(int F4)
{
    // создание переменной result_f7, которая б
    int result_F7 = F4 - 32;
    // возвращение итоговой переменной result_F7
    return result_F7;
}
```

Рисунок 21 - Функция F7(int F4)

В качестве параметра передаётся численное значение функции F4. Внутри функции создаётся результирующая переменная result_F7, которой присваивается значение некой математической операции с передаваемым параметром.

8) F8(int F5, int F6, int F7). Её содержимое продемонстрировано на рисунке 22.

```
public int F8(int F5, int F6, int F7)
{
    // инициализация переменной result_F7, которая будет
    // использоваться для хранения результата умножения F5, F6, F7 */
    int result_F8 = F5 * (F7 - F6);
    // возвращение итоговой переменной result_F8
    return result_F8;
}
```

Рисунок 22 - Функция F8(int F5, int F6, int F7)

В качестве параметра передаются численные значения функций F5, F6, F7. Внутри функции создаётся результирующая переменная result_F8, которой присваивается значение неких математических операций с передаваемыми параметрами.

Задачи

1) Task_A(). Содержимое метода продемонстрировано на рисунке 23.

```

public void Task_A()
{
    // Инициализация переменной startTime, которая хранит в себе результат
    long startTime = Information_of_Thread("A", "UI-тред");
    // Приостановление выполнения текущего потока на 1000 мс
    System.Threading.Thread.Sleep(1000);
    // Вывод сообщения
    UpdateText("Генерируем значение M \n");
    // Генерация двумерного массива M (матрицы)
    Random_matrix(n);
    // Вывод сообщения
    UpdateText("M сгенерирована \n");
    // Вывод сообщения
    UpdateText("Генерируем значение R \n");
    // Генерация одномерного массива R
    R_generation(n);
    // Вывод сообщения
    UpdateText("R сгенерирована \n\n");
    // Вызов функции Information_Of_Finished_Task, которая выводит инфо
    Information_Of_Finished_Task("A", startTime);
    // Создание и запуск 3 новых потоков (Thread_B, Thread_C, и Thread_D)
    Thread_B = new Thread(() => Task_B("A"));
    Thread_B.Start();
    Thread_C = new Thread(() => Task_C("A"));
    Thread_C.Start();
    Thread_D = new Thread(() => Task_D("A"));
    Thread_D.Start();
    // Обновление значения ProgressBar для задачи A (значение в 100%)
    UpdateProgressBar(progressBar1, 100);
}

```

Рисунок 23 — Метод Task_A()

Данный метод описывает задачу A из индивидуального варианта. Для начала переменной startTime передаётся значение ранее рассмотренного метода Information_of_Thread – сообщения со всей необходимой информацией о начале работы задачи. Затем согласно индивидуальному варианту задача приостанавливается на 1000 мс. После этого в текстовое поле выводится значение ранее рассмотренной функции UpdateText – сообщение о том, что начинается генерация матрицы M. Затем вызывается соответствующий метод Random_matrix(n). после этого повторно вызывается метод UpdateText –

в текстовое поле выводится сообщение о том, что матрица сгенерирована. Аналогично проходит и генерация массива R. Затем вызывается ранее рассмотренный метод `Information_Of_Finished_Task`, с помощью которого выводится соответствующая информация о завершённой задаче.

Далее запускается 3 новых потока в соответствии с приведённым в индивидуальном варианте графом и обновляются значения необходимых прогресс-баров.

2) `Task_B()`. Содержимое метода продемонстрировано на рисунке 24.

```
public void Task_B(String ParentThread)
{
    // Инициализация переменной startTime, которая хранит в себе результат вычисления
    long startTime = Information_of_Thread("B", ParentThread);
    // Установка времени выполнения текущего потока на 1000 мс
    System.Threading.Thread.Sleep(1000);
    // Вывод сообщения
    UpdateText(" Исполняем функцию f1");
    // Вычисление результирующей переменной result_F1 значение функции F1
    result_F1 = F1(M, R);
    // Вывод сообщения с результатом функции F1
    UpdateText(" Значение F1: " + result_F1.ToString());
    // Вызов функции Information_Of_Finished_Task, которая выводит информацию
    Information_Of_Finished_Task("B", startTime);
    // Блокировка потоков D и C, пока семафор не станет доступен
    semaphore.WaitOne();
    // Увеличение количества доступных разрешений в семафоре
    semaphore.Release();
    // Проверка: не завершились ли другие задачи (Thread_D и Thread_C)?
    if ((!Thread_D.IsAlive) & (!Thread_C.IsAlive))
    {
        // Если задачи завершились, то запускаются следующие 2 задачи (Thread_E, Thread_F)
        Thread_E = new Thread(() => Task_E("B"));
        Thread_E.Start();
        Thread_F = new Thread(() => Task_F("B"));
        Thread_F.Start();
        // Обновление ProgressBar по задачам B, C, D соответственно
        UpdateProgressBar(progressBar2, 100);
        UpdateProgressBar(progressBar3, 100);
        UpdateProgressBar(progressBar4, 100);
    }
}
```

Рисунок 24 — Метод `Task_B()`

Данный метод описывает задачу В из индивидуального варианта. В качестве параметра передаётся имя родительского потока — потока, благодаря которому был запущен текущий. Для начала переменной `startTime` передаётся значение ранее рассмотренного метода `Information_of_Thread` – сообщения со всей необходимой информацией о начале работы задачи. Затем согласно индивидуальному варианту задача приостанавливается на 1000 мс. После этого в текстовое поле выводится значение ранее рассмотренной функции `UpdateText` – сообщение о том, что выполняется необходимая функция. Далее созданной переменной `result_F1` присваивается значение ранее рассмотренного метода `F1(M,R)` с последующим выводом сообщения о том, что было получено такое-то значение функции `F1` и что завершилась текущая задача со всей необходимой подробной информацией.

После этого с помощью созданного семафора реализуется блокировка потоков, которые идут в параллель с текущим, пока не будет открыт доступ для предоставления разрешения и последующее увеличение количества доступных разрешений.

Далее проводится проверка: не завершились ли другие параллельные задачи? Если все параллельные задачи завершены, то запускаются следующие задачи согласно графу из индивидуального варианта и обновляются значения соответствующих прогресс-баров.\

Аналогично работают и методы `Task_C`, `Task_D`, `Task_E`, `Task_F`, `Task_G`, `Task_H`.

3) `Task_K()`. Содержимое метода продемонстрировано на рисунке 25.

```

public void Task_K(String ParentThread)
{
    // Инициализация переменной startTime, которая хранит в себе результат вы
    long startTime = Information_of_Thread("K", ParentThread);
    // Установка выполнения текущего потока на 1000 мс
    System.Threading.Thread.Sleep(1000);
    // Вывод сообщения
    UpdateText(" Исполняем функцию f8");
    // Наименование результирующей переменной result_F8 значение функции F
    result_F8 = F8(result_F5, result_F6, result_F7);
    // Вывод сообщения с результатом функции F8
    UpdateText(" Значение F8: " + result_F8.ToString());
    // Функция Information_Of_Finished_Task, которая выводит информа
    Information_Of_Finished_Task("K", startTime);
    // Обновление ProgressBar по задаче K
    UpdateProgressBar(progressBar9, 100);
}

```

Рисунок 25 — Метод Task_K()

Содержимое метода имеет то же строение и логику, что и в прошлом методе, за исключением того, что в последней задаче — собственно, K — не используется семафор за ненужностью. Поэтому метод лишён этой части и после вывода сообщения о завершении задачи идёт просто обновление прогресс-бара.

Иные структуры

В данном блоке рассмотрим иные структуры, использовавшиеся для реализации поставленной задачи.

1) Класс Form1. Часть кода с его упоминанием продемонстрирована на рисунке 26.

```

// Ссылка: S
public partial class Form1 : Form
{

```

Рисунок 26 — Класс Form1

Именно внутри этого класса и содержатся все описанные ранее методы, функции, части интерфейса и переменные. Он является наследником класса Form, который является частью пространства System.Windows.Forms.

2) Переменные

Были инициализированы такие переменные, как:

- `int[,] M` – численный двумерный массив `M`
- `int[] R` – численный одномерный массив `R`
- `int n = 4` – размерность массивов, значение было выбрано разработчиком \
- `long ExecuteStartTime` – переменная, обозначающая время начала работы задачи
- `int FinishedTasksCounter` – переменная, которая обозначает количество завершённых задач

3) Потоки

Были объявлены такие потоки, как:

- `Thread Thread_A` – Поток соответствует задаче `A` из индивидуального варианта
- `Thread Thread_B` – Поток соответствует задаче `B` из индивидуального варианта
- `Thread Thread_C` – Поток соответствует задаче `C` из индивидуального варианта
- `Thread Thread_D` – Поток соответствует задаче `D` из индивидуального варианта
- `Thread Thread_E` – Поток соответствует задаче `E` из индивидуального варианта

- Thread Thread_F – Поток соответствует задаче F из индивидуального варианта
- Thread Thread_G – Поток соответствует задаче G из индивидуального варианта
- Thread Thread_H – Поток соответствует задаче H из индивидуального варианта
- Thread Thread_K – Поток соответствует задаче K из индивидуального варианта

4) Результирующие переменные:

- int result_F1 – переменная, хранящая в себе результат функции F1
- int result_F2 – переменная, хранящая в себе результат функции F2
- int result_F3 – переменная, хранящая в себе результат функции F3
- int result_F4 – переменная, хранящая в себе результат функции F4
- int result_F5 – переменная, хранящая в себе результат функции F5
- int result_F6 – переменная, хранящая в себе результат функции F6
- int result_F7 – переменная, хранящая в себе результат функции F7
- int result_F8 – переменная, хранящая в себе результат функции F8

5) Семафор

static Semaphore semaphore = new Semaphore(3, 3) – семафор с таким ограничением: до 3 потоков могут одновременно получить доступ к защищённому ресурсу, другие потоки должны будут ждать, пока разрешения не «освободятся».

6) Объект класса Random – rng – служит для генерации псевдослучайных чисел в заданном диапазоне

Код программы

```
/*Подключение пространств имён. */

/* System содержит основные классы и типы данных. Включает в себя
* также и операции ввода-вывода, работу с исключениями и управление памятью. */

using System;

/* System.Collections.Generic предоставляет общие коллекции (структуры данных) и интерфейсы
(списки,
* словари, множества) */

using System.Collections.Generic;

/* System.ComponentModel содержит классы и интерфейсы, связанные с компонентной моделью .NET
*/

using System.ComponentModel;

/* System.Data предоставляет классы и интерфейсы для работы с данными и базами данных */

using System.Data;

/* System.Drawing предоставляет классы для работы с графикой, изображениями и рисованием */

using System.Drawing;

/* LINQ (Language-Integrated Query) — это часть .NET Framework, предоставляющая возможность
* выполнения запросов к данным непосредственно в коде */

using System.Linq;

/* System.Text предоставляет классы для работы с символьными данными, кодировками и
* текстовыми операциями */

using System.Text;

/* System.Threading предоставляет классы и интерфейсы для работы с потоками выполнения
* (иначе threads) в многозадачных приложениях */

using System.Threading;

/* System.Threading.Tasks предоставляет классы и интерфейсы для работы с параллельным
* программированием и асинхронными операциями */

using System.Threading.Tasks;

/* System.Windows.Forms предоставляет классы и компоненты для создания графического
```

```

* пользовательского интерфейса (иначе GUI) в приложениях Windows */

using System.Windows.Forms;


// Объявление пространства имён
namespace lb_2_S5_Tegai_4143
{
/* Объявление класса Form1, который наследуется от класса Form,
* который является частью пространства System.Windows.Forms */
    public partial class Form1 : Form
    {
// Объявление численного двумерного массива M
        int[, ] M;

/* Объявление переменной n, которая обозначает размерность массива.
* Значение, выбранное для n, выбиралось разработчиком (рекомендуемое значение:  $3 \leq n \leq 6$ ) */
        int n = 4;

// Объявление одномерного массива R
        int[] R;

// Объявление переменной ExecuteStartTime, которая обозначает время начала работы задачи
        long ExecuteStartTime;

// Объявление переменной FinishedTasksCounter, которая обозначает количество завершённых задач
        int FinishedTasksCounter;


// Объявление потоков Thread_A - Thread_K
        Thread Thread_A;
        Thread Thread_B;
        Thread Thread_C;
        Thread Thread_D;
        Thread Thread_E;
        Thread Thread_F;
        Thread Thread_G;
        Thread Thread_H;
        Thread Thread_K;

```

// Объявление переменных result_F1 - result_F8, которые хранят в себе результаты функций F1 - F8 соответственно

int result_F1;

int result_F2;

int result_F3;

int result_F4;

int result_F5;

int result_F6;

int result_F7;

int result_F8;

/* Объявление конструктора класса Form1. InitializeComponent инициализирует компоненты формы,

* такие как кнопки, текстовые поля и другие элементы управления */

public Form1()

{

InitializeComponent();

}

/* Создание статического объекта семафора для задач.

*

* Семафор это многозадачный синхронизационный механизм, который позволяет ограничивать доступ к

* определенному ресурсу или критической секции в многозадачном окружении.

* Семафоры используются для координации работы множества потоков или процессов в условиях

* разделяемого доступа к ресурсам.

*

* Первое число в скобках обозначает начальное количество разрешений, которые доступны семафору.

* Второе число обозначает максимальное количество разрешений, которые может иметь семафор.

*

* Например, запись static Semaphore semaphore = new Semaphore(3, 3) обозначает, что

* до 3 потоков могут одновременно получить доступ к защищенному ресурсу. Другие потоки

* должны будут ждать, пока разрешения не "освободятся" */

```

static Semaphore semaphore = new Semaphore(3, 3);

// Создание rng - объекта класса Random, который служит для генерации псевдослучайных чисел

Random rng = new Random();

// МЕТОДЫ

/* Создание метода Random_matrix для генерации матрицы M размером n x n, которая состоит из
случайно
* подобранных чисел */

public void Random_matrix(int n)
{
// Объявление матрицы M, имеющей размерность n x n

M = new int[n, n];

// Прохождение по столбцам

for (int i = 0; i < n; i++)
{
// Прохождение по строкам

for (int j = 0; j < n; j++)
{
// Заполнение случайным числом от 1 до 50 числа матрицы, стоящим в i-м столбце j-й строки

M[i, j] = rng.Next(51);

}

}

}

// Метод R_generation, с помощью которого генерируется одномерный массив R, размером n

public void R_generation(int n)
{
// Объявление переменной R - массива с размером n

R = new int[n];

// Прохождение по столбцам

for (int i = 0; i < n; i++)

```

```

    {

// Присваивание каждому элементу массива случайное число от 1 до 50

        R[i] = rng.Next(51);

    }

}

// Метод UpdateText, который обновляет текстовое поле textBox1 в графическом интерфейсе.

    public void UpdateText(String what)

    {

/* Если вызов произошел из потока, инвокируем в главном потоке.

* В общем, это проверка на то, выполняется ли текущий код в потоке, в котором был создан

* элемент управления - Form1. Invoke используется для безопасного доступа к элементам управления
*/

        if (InvokeRequired)

        {

/* Invoke гарантирует, что UpdateText будет выполнен в потоке, в котором был создан элемент
управления

this */

            this.Invoke(new Action<string>(UpdateText), new object[] { what });

            return;

        }

// Если текстовое поле пустое, то перенос строки в начало не производится

        String pre = "\r\n";

        if (textBox1.Text == "")

        {

            pre = "";

        }

// Обновление текстового поля с учетом вызова из разных потоков

        textBox1.Text += pre + what.ToString();

    }

/* Объявление метода UpdateProgressBar, который обновляет значение ProgressBar

(ProgressBar) в графическом интерфейсе.*/

```

```

private void UpdateProgressBar(ProgressBar progressBar, int value)
{
    if (progressBar.InvokeRequired)
    {
        /* По той же логике, как и в методе UpdateText, если вызов происходит из потока, отличного от
        главного,
        *в целях безопасности используется Invoke */
        progressBar.Invoke(new Action<ProgressBar, int>(UpdateProgressBar), progressBar, value);
        return;
    }

    // Установление нового значения для ProgressBar в соответствии с переданным параметром value
    progressBar.Value = value;
}

// Объявление метода Timestamp, который используется для измерения времени выполнения задачи
public long Timestamp()
{
    // Возвращение текущей метки времени (в мс)
    return DateTimeOffset.UtcNow.ToUnixTimeMilliseconds();
}

// Объявление метода Information_of_Thread, который предназначен для записи информации о запуске
задачи
public long Information_of_Thread(String ActiveThread, String ParentThread)
{
    // Запись информации о запуске задачи (какая задача запущена, с помощью какой задачи она
    была запущена и в какой момент времени (в мс)
    UpdateText(" Задача " + ActiveThread + " запущена с помощью задачи " + ParentThread + " в " +
    (Timestamp() - ExecuteStartTime).ToString() + " мс ");
    // Возвращение значения метода Timestamp (в мс)
    return Timestamp();
}

```


// Объявление метода Information_Of_Finished_Task, который предназначен для записи информации о завершении задачи

```
public void Information_Of_Finished_Task(String ActiveThread, long StartTime)
{
```

// Если вызов происходит из потока, отличного от главного, в целях безопасности используется Invoke

```
    if (InvokeRequired)
    {
        this.Invoke(new Action<string, long>(Information_Of_Finished_Task), new object[]
        { ActiveThread, StartTime });
        return;
    }
```

// Присваивание переменной NowTime значения текущего времени (в мс)

```
    long NowTime = Timestamp();
```

// Вычисление времени выполнения задачи

```
    String ExecTime = (NowTime - StartTime).ToString();
```

// Запись информации о завершении задачи

```
    UpdateText(" Задача " + ActiveThread + " завершилась.\r\nЕё время выполнения: " + ExecTime +
" мс.\r\nВремя окончания: " + (NowTime - ExecuteStartTime).ToString() + " мс \r\n");
}
```

// ФУНКЦИИ

// Объявление функции F1 с параметрами M (двумерный целочисленный массив) и R (одномерный целочисленный массив)

```
public int F1(int[,] M, int[] R)
{
```

// Объявление переменной summa, которая будет хранить в себе значение суммы различных математических вычислений

```
    int summa = 0;
```

// Объявление 1-го элемента массива R - element

```
    int element = R[1];
```

/* Объявление переменной Intermediate_Summa - промежуточная сумма.

* Эта переменная будет хранить в себе значение некой математической операции */

```
    int Intermediate_Summa = 0;
```

// Прохождение по 0-му столбцу массива M (матрицы)

```

        for (int i = 0; i < 1; i++)
        {
// Прохождение по строкам массива М (матрицы)
            for (int j = 0; j < n; j++)
            {
// К каждому рассматриваемому элементу матрицы прибавляется значение 1-го элемента массива R
                Intermediate_Summa = M[i, j] + element;
// Затем это значение добавляется (прибавляется) к итоговой сумме
                summa += Intermediate_Summa;
            }
        }

// Возвращение итоговой суммы
        return summa;
    }

// Объявление функции F2 с параметрами М (двумерный целочисленный массив) и R (одномерный
целочисленный массив)
    public int F2(int[,] M, int[] R)
    {
// Объявление переменной subtraction, которая будет хранить в себе значение вычитаний в ходе
различных математических вычислений
        int subtraction = 100;

// Объявление 2-го элемента массива R - element
        int element = R[2];

/* Объявление переменной Intermediate_subtraction - промежуточная переменная, хранящая в себе
результат вычитания */
        int Intermediate_subtraction = 0;

// Прохождение по 1-му столбцу массива М матрицы)
        for (int i = 1; i < 2; i++)
        {
// Прохождение по строкам массива М (матрицы)
            for (int j = 0; j < n; j++)
            {

```

```

// К каждому рассматриваемому элементу матрицы прибавляется значение 2-го элемента массива R
    Intermediate_substraction = M[i, j] + element;

// Затем это значение вычитается из переменной substraction
    substraction -= Intermediate_substraction;

    }

}

// Возвращение переменной substraction
    return substraction;

}

// Объявление функции F3 с параметрами M (двумерный целочисленный массив) и R (одномерный
целочисленный массив)

    public int F3(int[,] M, int[] R)

    {

// Объявление переменной summa, которая будет хранить в себе значение суммы различных
математических вычислений

        int summa = 0;

// Объявление 3-го элемента массива R - element

        int element = R[3];

/* Объявление переменной Intermediate_Summa - промежуточная сумма.

* Эта переменная будет хранить в себе значение некой математической операции */

        int Intermediate_Summa = 0;

// Прохождение по всем диагональным элементам матрицы M

        for (int i = 0; i < 4; i++)

        {

// К каждому рассматриваемому элементу матрицы прибавляется значение 3-го элемента массива R в
квадрате

            Intermediate_Summa = M[i, i] + element^2;

// Затем это значение добавляется (прибавляется) к итоговой сумме

            summa += Intermediate_Summa;

        }

// Возвращение итоговой суммы

        return summa;

```

```
}
```

```
// Объявление функции F4 с параметрами-функциями F1, F2, F3
```

```
public int F4(int F1, int F2, int F3)
```

```
{
```

```
/* Объявление переменной result_F4, которая будет хранить в себе значение некоего математического действия
```

```
* со значениями функций F1, F2, F3*/
```

```
int result_F4 = F2 + F1 - F3;
```

```
// Возвращение итоговой переменной result_F4
```

```
return result_F4;
```

```
}
```

```
// Объявление функции F5 с параметрами-функциями F1, F2, F3
```

```
public int F5(int F1, int F2, int F3)
```

```
{
```

```
/* Объявление переменной result_f5, которая будет хранить в себе значение некоего математического действия
```

```
* со значениями функций F1, F2, F3*/
```

```
int result_F5 = F1 - F2 + F3;
```

```
// Возвращение итоговой переменной result_f5
```

```
return result_F5;
```

```
}
```

```
// Объявление функции F6 с параметром-функцией F4
```

```
public int F6(int F4)
```

```
{
```

```
// Объявление переменной result_F6, которая будет хранить в себе значение суммы значения функции F4 и числа 14
```

```
int result_F6 = F4 + 14;
```

```
// Возвращение итоговой переменной result_f6
```

```
return result_F6;
```

```
}
```

```
// Объявление функции F7 с параметром-функцией F4
```

```
public int F7(int F4)
```

```
{
```

```
// Объявление переменной result_f7, которая будет хранить в себе значение вычитания из значения функции f4 числа 32
```

```
int result_F7 = F4 - 32;
```

```
// Возвращение итоговой переменной result_F7
```

```
return result_F7;
```

```
}
```

```
// Объявление функции F8 с параметрами-функциями F5, F6, F7
```

```
public int F8(int F5, int F6, int F7)
```

```
{
```

```
/* Объявление переменной result_F7, которая будет хранить в себе значение некоего математического действия со значениями
```

```
* функций F5, F6, F7 */
```

```
int result_F8 = F5 *(F7 - F6);
```

```
// Возвращение итоговой переменной result_F8
```

```
return result_F8;
```

```
}
```

```
// ЗАДАЧИ
```

```
//Объявление задачи A, которая представляет собой задачу, запущенную в основном пользовательском (UI) потоке
```

```
public void Task_A()
```

```
{
```

```
// Объявление переменной startTime, которая хранит в себе результат выполнения функции Information_of_Thread (начало работы)
```

```
long startTime = Information_of_Thread("A", "UI-тред");
```

```
// Приостановление выполнения текущего потока на 1000 мс
```

```
System.Threading.Thread.Sleep(1000);
```

```
// Вывод сообщения
```

```
UpdateText("Генерируем значение M \n");
```

```
// Генерация двумерного массива M (матрицы)
```

```

Random_matrix(n);

// Вывод сообщения

UpdateText("М сгенерирована \n");

// Вывод сообщения

UpdateText("Генерируем значение R \n");

// Генерация одномерного массива R

R_generation(n);

// Вывод сообщения

UpdateText("R сгенерирована \n\n");

// Вызов функции Information_Of_Finished_Task, которая выводит информацию о завершении задачи

Information_Of_Finished_Task("A", startTime);

//Создание и запуск 3 новых потоков (Thread_B, Thread_C, и Thread_D), каждый из которых
представляет собой отдельную задачу

Thread_B = new Thread(() => Task_B("A"));

Thread_B.Start();

Thread_C = new Thread(() => Task_C("A"));

Thread_C.Start();

Thread_D = new Thread(() => Task_D("A"));

Thread_D.Start();

// Обновление значения ProgressBar для задачи A (значение в 100%)

UpdateProgressBar(progressBar1, 100);

}

//Объявление задачи B

public void Task_B(String ParentThread)

{

// Объявление переменной startTime, которая хранит в себе результат выполнения функции
Information_of_Thread (начало работы)

long startTime = Information_of_Thread("B", ParentThread);

// Приостановление выполнения текущего потока на 1000 мс

System.Threading.Thread.Sleep(1000);

// Вывод сообщения

UpdateText(" Исполняем функцию F1");

```

```

// Присваивание результирующей переменной result_F1 значение функции F1
    result_F1 = F1(M, R);

// Вывод сообщения с результатом функции F1
    UpdateText(" Значение F1: " + result_F1.ToString());

// Вызов функции Information_Of_Finished_Task, которая выводит информацию о завершении задачи
    Information_Of_Finished_Task("B", startTime);

    // Блокировка потоков D и C, пока семафор не станет доступным для предоставления
    разрешения
    semaphore.WaitOne();

    // Увеличение количество доступных разрешений в семафоре
    semaphore.Release();

// Проверка: не завершились ли другие задачи (Thread_D и Thread_C)?
    if ((!Thread_D.IsAlive) & (!Thread_C.IsAlive))
    {
//Если обе завершились, то запускаются следующие 2 задачи (Thread_E, Thread_F)
        Thread_E = new Thread(() => Task_E("B"));
        Thread_E.Start();
        Thread_F = new Thread(() => Task_F("B"));
        Thread_F.Start();

//Обновление ProgressBar по задачам B, C, D соответственно
        UpdateProgressBar(progressBar2, 100);
        UpdateProgressBar(progressBar3, 100);
        UpdateProgressBar(progressBar4, 100);
    }
}

// Объявление задачи C
    public void Task_C(String ParentThread)
    {
// Объявление переменной startTime, которая хранит в себе результат выполнения функции
Information_of_Thread (начало работы)
        long startTime = Information_of_Thread("C", ParentThread);

// Приостановление выполнения текущего потока на 1000 мс

```

```

        System.Threading.Thread.Sleep(1000);

        // Вывод сообщения

        UpdateText(" Исполняем функцию F2");

    // Присваивание результирующей переменной result_F2 значение функции F2

        result_F2 = F2(M, R);

        // Вывод сообщения с результатом функции F2

        UpdateText(" Значение F2: " + result_F2.ToString());

    // Вызов функции Information_Of_Finished_Task, которая выводит информацию о завершении задачи

        Information_Of_Finished_Task("C", startTime);

    // Блокировка потоков D и C, пока семафор не станет доступным для предоставления разрешения

        semaphore.WaitOne();

    // Увеличение количество доступных разрешений в семафоре

        semaphore.Release();

    // Проверка: не завершились ли другие задачи (Thread_D и Thread_B)?

        if ((!Thread_D.IsAlive) & (!Thread_B.IsAlive))

        {

    //Если обе завершились, то запускаются следующие 2 задачи (Thread_E, Thread_F)

            Thread_E = new Thread(() => Task_E("C"));

            Thread_E.Start();

            Thread_F = new Thread(() => Task_F("C"));

            Thread_F.Start();

    //Обновление ProgressBar по задачам C, B, D соответственно

            UpdateProgressBar(progressBar3, 100);

            UpdateProgressBar(progressBar2, 100);

            UpdateProgressBar(progressBar4, 100);

        }

    }

    // Объявление задачи D

    public void Task_D(String ParentThread)

    {

    // Объявление переменной startTime, которая хранит в себе результат выполнения функции
    Information_of_Thread (начало работы)

```



```

        long startTime = Information_of_Thread("D", ParentThread);

// Приостановление выполнения текущего потока на 1000 мс

        System.Threading.Thread.Sleep(1000);

// Вывод сообщения

        UpdateText(" Исполняем функцию F3");

// Присваивание результирующей переменной result_F3 значение функции F3

        result_F3 = F3(M, R);

// Вывод сообщения с результатом функции F3

        UpdateText(" Значение F3: " + result_F3.ToString());

// Вызов функции Information_Of_Finished_Task, которая выводит информацию о завершении задачи

        Information_Of_Finished_Task("D", startTime);

// Блокировка потоков B и C, пока семафор не станет доступным для предоставления разрешения

        semaphore.WaitOne();

// Увеличение количество доступных разрешений в семафоре

        semaphore.Release();

// Проверка: не завершились ли другие задачи (Thread_B и Thread_C)?

        if ((!Thread_B.IsAlive) & (!Thread_C.IsAlive))

        {

//Если обе завершились, то запускаются следующие 2 задачи (Thread_E, Thread_F)

            Thread_E = new Thread(() => Task_E("D"));

            Thread_E.Start();

            Thread_F = new Thread(() => Task_F("D"));

            Thread_F.Start();

//Обновление ProgressBar по задачам D, B, C соответственно

            UpdateProgressBar(progressBar4, 100);

            UpdateProgressBar(progressBar2, 100);

            UpdateProgressBar(progressBar3, 100);

        }

    }

// Объявление задачи E

    public void Task_E(String ParentThread)

```

```

{
// Объявление переменной startTime, которая хранит в себе результат выполнения функции
Information_of_Thread (начало работы)

    long startTime = Information_of_Thread("E", ParentThread);

// Приостановление выполнения текущего потока на 1000 мс

    System.Threading.Thread.Sleep(1000);

// Вывод сообщения

    UpdateText(" Исполняем функцию F4");

// Присваивание результирующей переменной result_F4 значение функции F4

    result_F4 = F4(result_F1, result_F2, result_F3);

// Вывод сообщения с результатом функции F4

    UpdateText(" Значение F4: " + result_F4.ToString());

// Вызов функции Information_Of_Finished_Task, которая выводит информацию о завершении задачи

    Information_Of_Finished_Task("E", startTime);

// Блокировка потока F, пока семафор не станет доступным для предоставления разрешения

    semaphore.WaitOne();

// Увеличение количество доступных разрешений в семафоре

    semaphore.Release();

// Проверка: не завершились ли другие задачи (Thread_B, Thread_C и Thread_D)?

    if ((!Thread_B.IsAlive) & (!Thread_C.IsAlive) & (!Thread_D.IsAlive))

    {

//Если все 3 завершились, то запускаются следующие 3 задачи (Thread_G, Thread_H и Thread_F)

        Thread_G = new Thread(() => Task_G("E"));

        Thread_G.Start();

        Thread_H = new Thread(() => Task_H("E"));

        Thread_H.Start();

        Thread_F = new Thread(() => Task_F("E"));

        Thread_F.Start();

    }

/* Обновление ProgressBar по задачам E, F соответственно

* Здесь ProgressBar задачи F присваивается значение в 50% потому, что на момент завершения задачи
E
задача F завершится лишь наполовину, то бишь на 50%*/

```

```

        UpdateProgressBar(progressBar5, 100);

        UpdateProgressBar(progressBar6, 50);
    }

// Объявление задачи F

    public void Task_F(String ParentThread)
    {
        // Объявление переменной startTime, которая хранит в себе результат выполнения функции
        Information_of_Thread (начало работы)

        long startTime = Information_of_Thread("F", ParentThread);

        // Приостановление выполнения текущего потока на 2000 мс

        System.Threading.Thread.Sleep(2000);

        // Вывод сообщения

        UpdateText(" Исполняем функцию F5");

        // Присваивание результирующей переменной result_F5 значение функции F5

        result_F5 = F5(result_F1, result_F2, result_F3);

        // Вывод сообщения с результатом функции F5

        UpdateText(" Значение F5: " + result_F5.ToString());

        // Вызов функции Information_Of_Finished_Task, которая выводит информацию о завершении задачи

        Information_Of_Finished_Task("F", startTime);

        // Блокировка потоков G и H, пока семафор не станет доступным для предоставления
        разрешения

        semaphore.WaitOne();

        // Увеличение количество доступных разрешений в семафоре

        semaphore.Release();

        // Проверка: не завершились ли другие задачи (Thread_G и Thread_H)?

        if ((!Thread_G.IsAlive) & (!Thread_H.IsAlive))
        {
            //Если обе завершились, то запускается следующая задача (Thread_K)

            Thread_K = new Thread(() => Task_K("F"));

            Thread_K.Start();
        }

        //Обновление ProgressBar по задачам G, H, K соответственно

```

```

        UpdateProgressBar(progressBar6, 100);

        UpdateProgressBar(progressBar7, 100);

        UpdateProgressBar(progressBar8, 100);
    }

    // Объявление задачи G

    public void Task_G(String ParentThread)
    {
        // Объявление переменной startTime, которая хранит в себе результат выполнения функции
        Information_of_Thread (начало работы)

        long startTime = Information_of_Thread("G", ParentThread);

        // Приостановление выполнения текущего потока на 1000 мс

        System.Threading.Thread.Sleep(1000);

        // Вывод сообщения

        UpdateText(" Исполняем функцию F6");

        // Присваивание результирующей переменной result_F6 значение функции F6

        result_F6 = F6(result_F4);

        // Вывод сообщения с результатом функции F6

        UpdateText(" Значение F6: " + result_F6.ToString());

        // Вызов функции Information_Of_Finished_Task, которая выводит информацию о завершении задачи

        Information_Of_Finished_Task("G", startTime);

        // Блокировка потоков H и F, пока семафор не станет доступным для предоставления
        разрешения

        semaphore.WaitOne();

        // Увеличение количество доступных разрешений в семафоре

        semaphore.Release();

        // Проверка: не завершились ли другие задачи (Thread_H и Thread_F)?

        if ((!Thread_H.IsAlive) & (!Thread_F.IsAlive))
        {
            //Если обе завершились, то запускается следующая задача (Thread_K)

            Thread_K = new Thread(() => Task_K("G"));

            Thread_K.Start();
        }
    }

```

//Обновление ProgressBar по задачам G, F, H соответственно

UpdateProgressBar(progressBar7, 100);

UpdateProgressBar(progressBar6, 100);

UpdateProgressBar(progressBar8, 100);

}

// Объявление задачи H

public void Task_H(String ParentThread)

{

// Объявление переменной startTime, которая хранит в себе результат выполнения функции Information_of_Thread (начало работы)

long startTime = Information_of_Thread("H", ParentThread);

// Приостановление выполнения текущего потока на 1000 мс

System.Threading.Thread.Sleep(1000);

// Вывод сообщения

UpdateText(" Исполняем функцию F7");

// Присваивание результирующей переменной result_F7 значение функции F7

result_F7 = F7(result_F4);

// Вывод сообщения с результатом функции F7

UpdateText(" Значение F7: " + result_F7.ToString());

// Вызов функции Information_Of_Finished_Task, которая выводит информацию о завершении задачи

Information_Of_Finished_Task("H", startTime);

// Блокировка потоков G и F, пока семафор не станет доступным для предоставления разрешения

semaphore.WaitOne();

// Увеличение количество доступных разрешений в семафоре

semaphore.Release();

/*semaphore_F.Release();*/

// Проверка: не завершились ли другие задачи (Thread_F и Thread_G)?

if ((!Thread_F.IsAlive) & (!Thread_G.IsAlive))

{

//Если обе завершились, то запускается следующая задача (Thread_K)

Thread_K = new Thread(() => Task_K("H"));

```

        Thread_K.Start();

    }

//Обновление ProgressBar по задачам H, F, G соответственно

    UpdateProgressBar(progressBar8, 100);

    UpdateProgressBar(progressBar6, 100);

    UpdateProgressBar(progressBar7, 100);

}

// Объявление задачи K

public void Task_K(String ParentThread)

{

// Объявление переменной startTime, которая хранит в себе результат выполнения функции
Information_of_Thread (начало работы)

    long startTime = Information_of_Thread("K", ParentThread);

// Приостановление выполнения текущего потока на 1000 мс

    System.Threading.Thread.Sleep(1000);

// Вывод сообщения

    UpdateText(" Исполняем функцию F8");

// Присваивание результирующей переменной result_F8 значение функции F8

    result_F8 = F8(result_F5, result_F6, result_F7);

// Вывод сообщения с результатом функции F8

    UpdateText(" Значение F8: " + result_F8.ToString());

// Вызов функции Information_Of_Finished_Task, которая выводит информацию о завершении задачи

    Information_Of_Finished_Task("K", startTime);

//Обновление ProgressBar по задаче K

    UpdateProgressBar(progressBar9, 100);

}

// ИНТЕРФЕЙС

// Объявление кнопки button1_Click (кнопка "Запустить задачу")

private void button1_Click(object sender, EventArgs e)

{

```

```
// Присваивание значения 0 переменной FinishedTasksCounter, которая отвечает за количество  
// завершенных задач
```

```
    FinishedTasksCounter = 0;
```

```
// Присваивание переменной ExecuteStartTime значения функции Timestamp (время начала выполнения)
```

```
    ExecuteStartTime = Timestamp();
```

```
// Создание и запуск потока Thread_A
```

```
    Thread_A = new Thread(new ThreadStart(Task_A));
```

```
    Thread_A.Start();
```

```
}
```

```
// Объявление кнопки button2_Click (кнопка "Очистить поле")
```

```
    private void button2_Click(object sender, EventArgs e)
```

```
    {
```

```
// Очищение текстового поля
```

```
        textBox1.Text = "";
```

```
// Обнуление всех ProgressBar
```

```
        progressBar1.Value = 0;
```

```
        progressBar2.Value = 0;
```

```
        progressBar3.Value = 0;
```

```
        progressBar4.Value = 0;
```

```
        progressBar5.Value = 0;
```

```
        progressBar6.Value = 0;
```

```
        progressBar7.Value = 0;
```

```
        progressBar8.Value = 0;
```

```
        progressBar9.Value = 0;
```

```
}
```

```
// Объявление названий для каждого ProgressBar
```

```
// "ЗАДАЧИ"
```

```
    private void label1_Click(object sender, EventArgs e)
```

```
    {
```

```
}
```

```
// Имя ProgressBar "A"
```

```
private void label2_Click(object sender, EventArgs e)
```

```
{
```

```
}
```

```
// Имя ProgressBar "B"
```

```
private void label3_Click(object sender, EventArgs e)
```

```
{
```

```
}
```

```
//Имя ProgressBar "C"
```

```
private void label4_Click(object sender, EventArgs e)
```

```
{
```

```
}
```

```
// Имя ProgressBar "D"
```

```
private void label5_Click(object sender, EventArgs e)
```

```
{
```

```
}
```

```
// Имя ProgressBar "E"
```

```
private void label6_Click(object sender, EventArgs e)
```

```
{
```

```
}
```



```
// Имя ProgressBar "F"
```

```
private void label7_Click(object sender, EventArgs e)

{

}

}
```

```
// Имя ProgressBar "G"
```

```
private void label8_Click(object sender, EventArgs e)

{

}

}
```

```
// Имя ProgressBar "H"
```

```
private void label9_Click(object sender, EventArgs e)

{

}

}
```

```
// Имя ProgressBar "K"
```

```
private void label10_Click(object sender, EventArgs e)

{

}

}
```

```
// PROGRESSBARS
```

```
// ProgressBar "A"
```

```
private void progressBar1_Click(object sender, EventArgs e)

{

}

}
```

```
// ProgressBar "B"
```

```
private void progressBar2_Click(object sender, EventArgs e)

{

}

// ProgressBar "C"

private void progressBar3_Click(object sender, EventArgs e)

{

}

// ProgressBar "D"

private void progressBar4_Click(object sender, EventArgs e)

{

}

// ProgressBar "E"

private void progressBar5_Click(object sender, EventArgs e)

{

}

// ProgressBar "F"

private void progressBar6_Click(object sender, EventArgs e)

{

}

// ProgressBar "G"

private void progressBar7_Click(object sender, EventArgs e)

{
```

```

    }

// ProgressBar "H"

    private void progressBar8_Click(object sender, EventArgs e)

    {

    }

// ProgressBar "K"

    private void progressBar9_Click(object sender, EventArgs e)

    {

    }

// Объявление текстового поля

    private void textBox1_TextChanged_1(object sender, EventArgs e)

    {

    }

// Объявление формы

    private void Form1_Load(object sender, EventArgs e)

    {

    }

}

```

Результат выполнения программы

Соответствующие результаты выполнения программы продемонстрированы на рисунках.

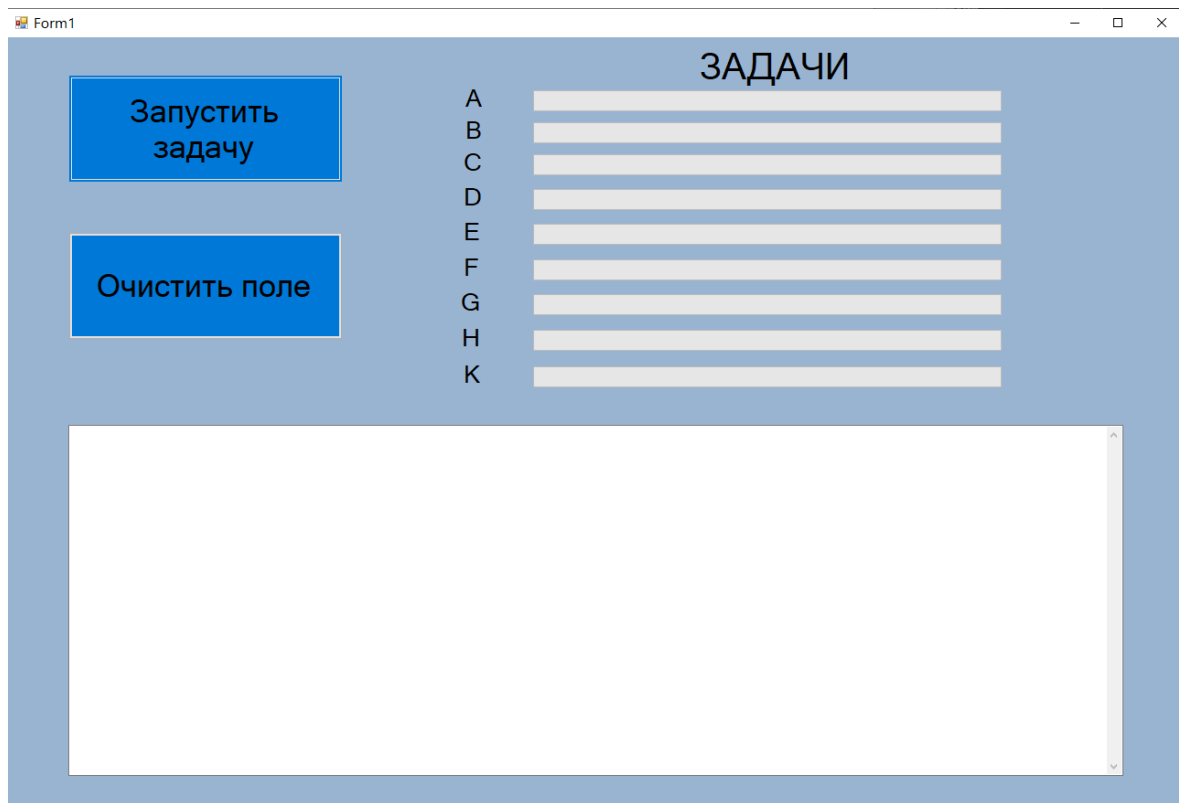


Рисунок 27 — Начальное окно приложения

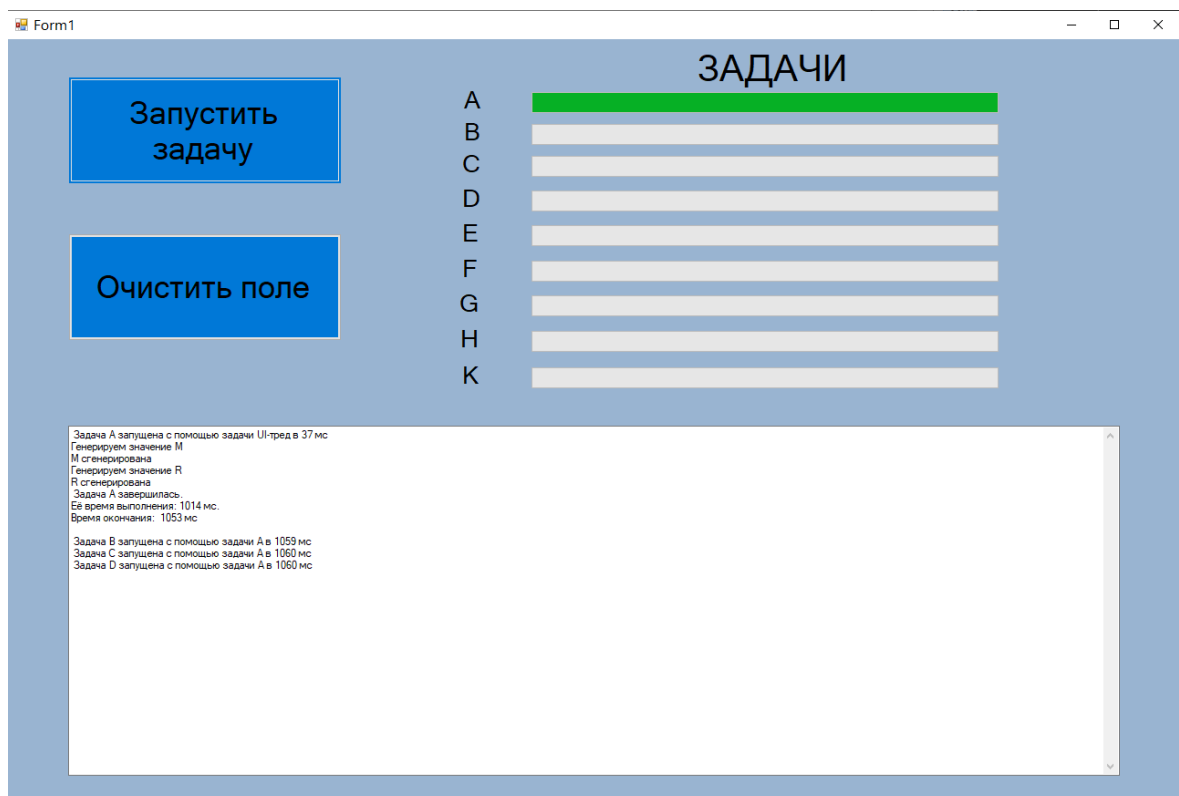


Рисунок 28 — Запуск задачи A с дальнейшим запуском следующих задач

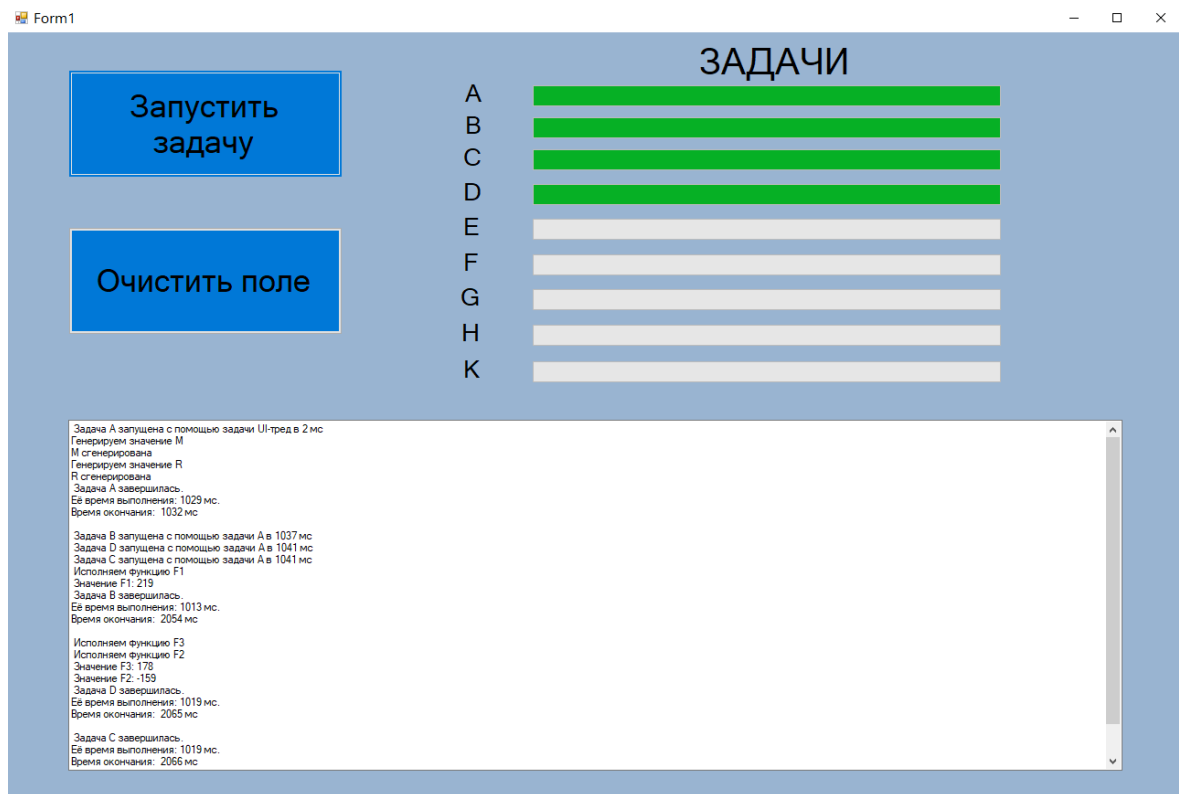


Рисунок 29 — Завершённые задачи B, C, D

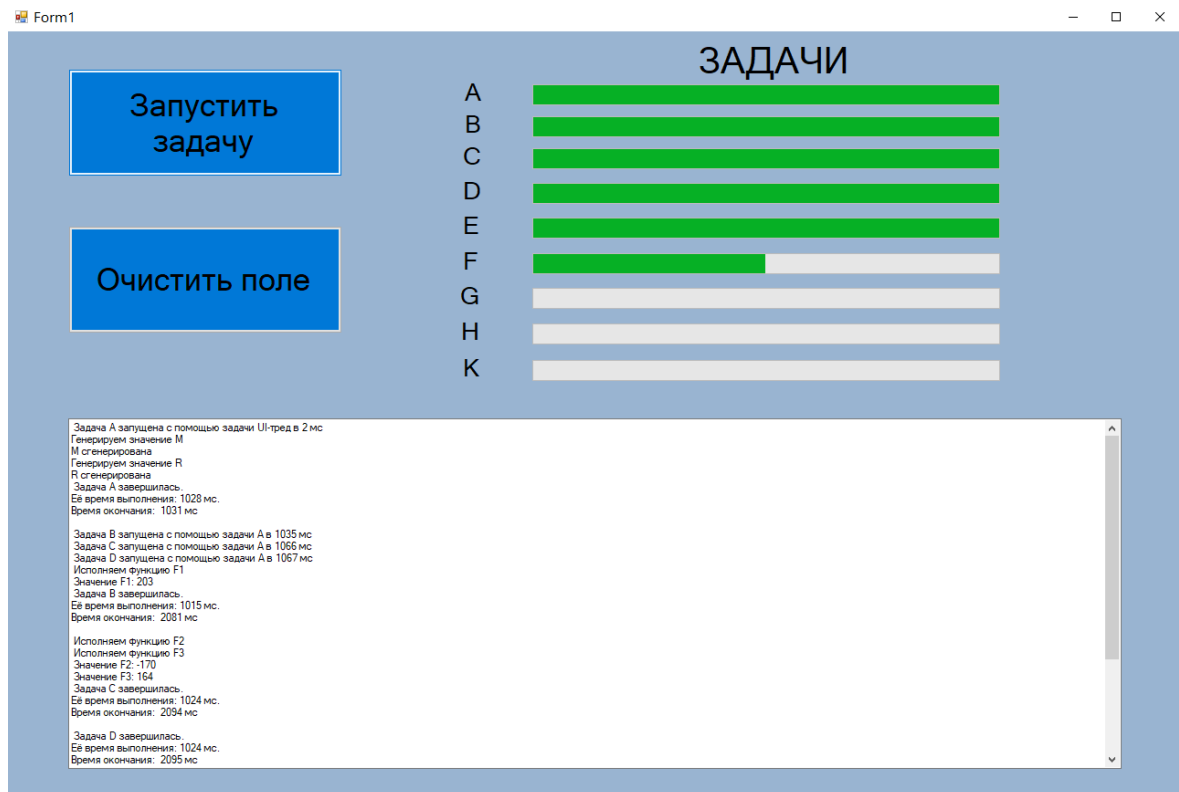


Рисунок 30 — Завершение задачи E и запуск задачи F

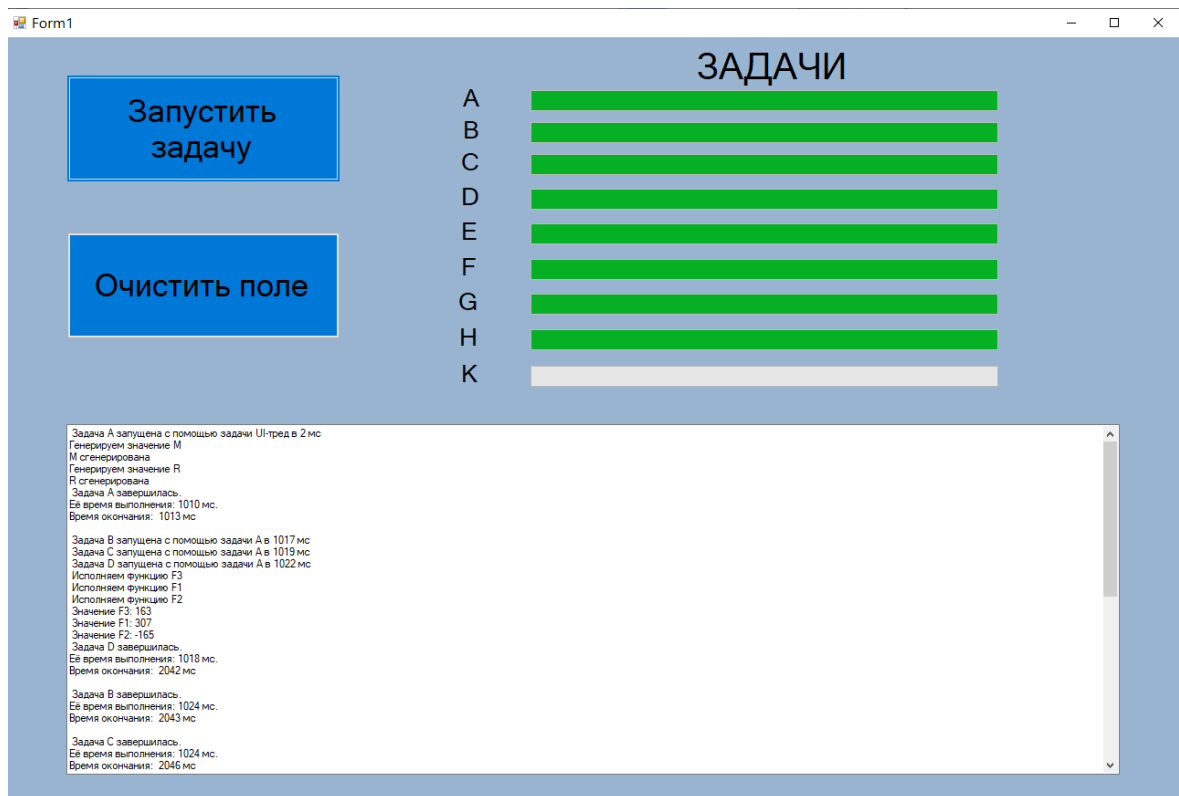


Рисунок 31 - Запуск и завершение задач G и H

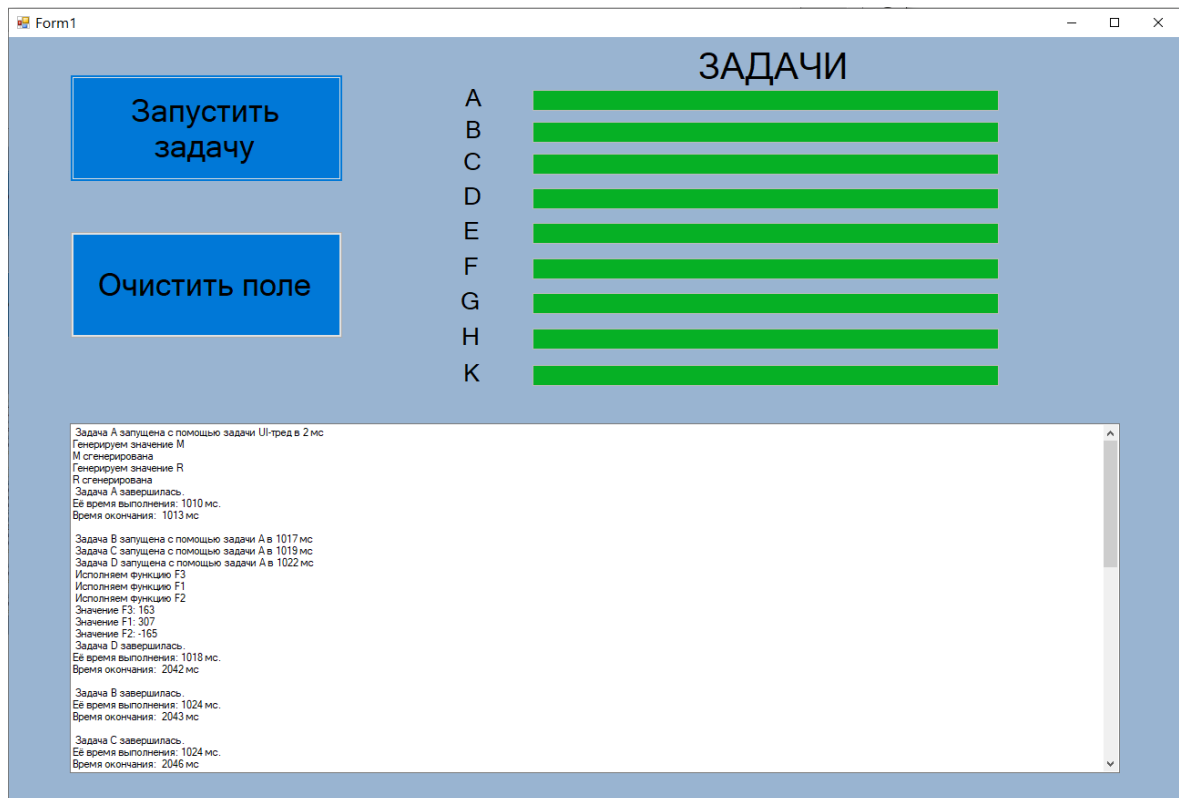


Рисунок 32 — Запуск и завершение задачи K

Содержимое текстового поля

Во время работы программы в текстовое поле добавлялась различная информация о запущенных и завершённых задачах, а также о результатах вычислений различных функций. Для удобства восприятия весь текст продемонстрирован ниже.

Задача А запущена с помощью задачи UI-тред в 2 мс

Генерируем значение М

М сгенерирована

Генерируем значение R

R сгенерирована

Задача А завершилась.

Её время выполнения: 1017 мс.

Время окончания: 1020 мс

Задача D запущена с помощью задачи А в 1060 мс

Задача В запущена с помощью задачи А в 1059 мс

Задача С запущена с помощью задачи А в 1061 мс

Исполняем функцию F3

Исполняем функцию F1

Исполняем функцию F2

Значение F3: 213

Значение F1: 257

Значение F2: -85

Задача D завершилась.

Её время выполнения: 1025 мс.

Время окончания: 2090 мс

Задача В завершилась.

Её время выполнения: 1025 мс.

Время окончания: 2092 мс

Задача С завершилась.

Её время выполнения: 1026 мс.

Время окончания: 2094 мс

Задача Е запущена с помощью задачи С в 2100 мс

Задача F запущена с помощью задачи С в 2103 мс

Исполняем функцию F4

Значение F4: -41

Задача Е завершилась.

Её время выполнения: 1022 мс.

Время окончания: 3126 мс

Задача G запущена с помощью задачи Е в 3131 мс

Задача H запущена с помощью задачи Е в 3134 мс

Задача F запущена с помощью задачи Е в 3138 мс

Исполняем функцию F5

Значение F5: 555

Задача F завершилась.

Её время выполнения: 2013 мс.

Время окончания: 4118 мс

Исполняем функцию F6

Значение F6: -27

Задача G завершилась.

Её время выполнения: 1011 мс.

Время окончания: 4149 мс

Исполняем функцию F7

Значение F7: -73

Задача H завершилась.

Её время выполнения: 1023 мс.

Время окончания: 4162 мс

Исполняем функцию F5

Значение F5: 555

Задача F завершилась.

Её время выполнения: 2020 мс.

Время окончания: 5160 мс

Задача K запущена с помощью задачи F в 5163 мс

Исполняем функцию F8

Значение F8: -25530

Задача K завершилась.

Её время выполнения: 1017 мс.

Время окончания: 6183 мс

Выводы

В данной лабораторной работе была написана и отлажена программа, которая реализует параллельное выполнение нескольких задач, каждая из которых решает некоторую заданную функцию согласно индивидуальному варианту.