Министерство науки и высшего образования

Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Российский государственный университет им. А.Н. Косыгина

(Технологии.Дизайн.Искусство)»

Кафедра автоматизированных систем обработки информации и управления

Лабораторная работа №3

Выполнила: Букша Кирилл Владимирович

Группа: МАГ-В-221

Вариант 4

Проверил: Кузьмина Тамара Михайловна

Москва 2021

Задание:

Создать консольное приложение, запускающее несколько потоков. Все потоки работают с двумя массивами - массивом A и массивом B, размерность у которых одинакова. Размеры массивов задаются пользователем и заполняется случайными числами. Исходные элементы массивов обязательно выводятся на экран. Массивы рассматриваются как записи векторов. Поэтому запись A + B читается, как сложение векторов, запись A*B, определяется как покоординатное произведение векторов, 2*A – умножение вектора на число, 2+A - прибавление числа 2 к каждому элементу массива A, т. е. программа выполняет действия над векторами и числами и получает в результате вектор. Каждое действие оформить как отдельную функцию.

Задание №1 выполнить все действия в одном потоке. Вывести ответы на экран.

Задание №2. Каждое действие выполнить в отдельном потоке, причем потоки должны работать «неторопливо», для этого нужно использовать функцию Sleep. Каждый поток должен выводить на экран результаты своих действий. Например, вычисляем выражение 2*A+B. Один поток вычисляет умножение вектора на число и выводит массив 2*A, второй вычисляет сумму векторов и выводит 2*A+B.

Сравнить результаты работы двух решений – однопоточного и многопоточного.

Вариант: (A + 7 * B) * 3

Решение:

Были написаны базовые функции для обработки вводимой информации, генерации и вывода данных:

```
private static void PrintData(IEnumerable<int> enumerable, string header)
{
    Console.Write($"{header}: ");
    foreach (var item in enumerable) Console.Write($"{item} ");
    Console.WriteLine();
}
private static (List<int>, List<int>) GenerateData(int size)
{
    var rnd = new Random();
    var vectorA = Enumerable.Range(0, size).Select(_ => rnd.Next(1, 20)).ToList();
    var vectorB = Enumerable.Range(0, size).Select(_ => rnd.Next(1, 20)).ToList();

    return (vectorA, vectorB);
}

private static int ReadUserInput()
{
    while (true)
    {
```

```
        Console.WriteLine("Enter q to exit.");
        Console.Write("Enter size of vectors: ");
        var input = Console.ReadLine();
        if (int.TryParse(input, out var size))
        {
            if (size <= 0)
            {
                Console.WriteLine("Size must be positive integer.");
                continue;
            }

            return size;
        }

        if (input == "q")
            throw new Exception();
    }

}
```

Далее были реализованы базовые математические операции в двух вариантах: быстром и медленном.

Быстрый вариант:

```
private static List<int> Sum(List<int> vector, int k)
{
    return vector.Select(n => n + k).ToList();
}

private static List<int> Sum(List<int> first, List<int> second)
{
    if (first.Count != second.Count)
        throw new ArgumentException("Pass lists with the same size!");

    var zippedLists = first.Zip(second, (f, s) => new { First = f, Second = s });

    return zippedLists.Select(item => item.First + item.Second).ToList();
}

private static List<int> Multiply(List<int> vector, int k)
{
    return vector.Select(n => n * k).ToList();
}

private static List<int> Multiply(List<int> first, List<int> second)
{
    if (first.Count != second.Count)
        throw new ArgumentException("Pass lists with the same size!");

    var zippedLists = first.Zip(second, (f, s) => new { First = f, Second = s });

    return zippedLists.Select(item => item.First * item.Second).ToList();

}
```

Медленный вариант:

```
private static List<int> SumSlow(List<int> vector, int k)
{
    Thread.Sleep(300);
```

```
    return vector.Select(n => n + k).ToList();
}

private static List<int> SumSlow(List<int> first, List<int> second)
{
    Thread.Sleep(300);
    if (first.Count != second.Count)
        throw new ArgumentException("Pass lists with the same size!");

    var zippedLists = first.Zip(second, (f, s) => new { First = f, Second = s });

    return zippedLists.Select(item => item.First + item.Second).ToList();
}

private static List<int> MultiplySlow(List<int> vector, int k)
{
    Thread.Sleep(300);
    return vector.Select(n => n * k).ToList();
}

private static List<int> MultiplySlow(List<int> first, List<int> second)
{
    Thread.Sleep(300);
    if (first.Count != second.Count)
        throw new ArgumentException("Pass lists with the same size!");

    var zippedLists = first.Zip(second, (f, s) => new { First = f, Second = s });

    return zippedLists.Select(item => item.First * item.Second).ToList();

}
```

Следующим шагом стала разработка основных функций, которые вычисляют результирующее выражение. Помимо указанных в задании вариантов были разработаны дополнительные, их назначение описано ниже. Итого были разработаны:

1) Однопоточный вариант без задержек;
2) Многопоточный вариант без задержек;
3) Многопоточный вариант с задержками;
4) Однопоточный вариант, модифицированный (Smart);
5) Многопоточный вариант, модифицированный (Smart).

Код приведен ниже:

```
private static List<int> CalculateFunctionSingleThread(List<int> vectorA, List<int>
vectorB)
{
    var multiplyResult = Multiply(vectorB, 7);
    var sumResult = Sum(vectorA, multiplyResult);
    var finalResult = Multiply(sumResult, 3);
    return finalResult;
}

private static List<int> CalculateFunctionMultipleThread(List<int> vectorA, List<int>
vectorB)
{
    var multiplyResult = Task.Run(() =>
    {
```

```
        var result = Multiply(vectorB, 7);
        PrintData(result, "Thread 1, results");
        return result;
    }).Result;

    var sumResult = Task.Run(() =>
    {
        var result = Sum(vectorA, multiplyResult);
        PrintData(result, "Thread 2, results");
        return result;
    }).Result;

    var finalResult = Task.Run(() =>
    {
        var result = Multiply(sumResult, 3);
        PrintData(result, "Thread 3, results");
        return result;
    }).Result;

    return finalResult;
}

private static List<int> CalculateFunctionMultipleThreadWithDelays(List<int> vectorA,
List<int> vectorB)
{
    var multiplyResult = Task.Run(() =>
    {
        var result = MultiplySlow(vectorB, 7);
        PrintData(result, "Thread 1, results");
        return result;
    }).Result;

    var sumResult = Task.Run(() =>
    {
        var result = SumSlow(vectorA, multiplyResult);
        PrintData(result, "Thread 2, results");
        return result;
    }).Result;

    var finalResult = Task.Run(() =>
    {
        var result = MultiplySlow(sumResult, 3);
        PrintData(result, "Thread 3, results");
        return result;
    }).Result;

    return finalResult;
}

private static List<int> CalculateFunctionSingleThreadSmart(List<int> vectorA, List<int>
vectorB)
{
    var firstMultiplyResult = MultiplySlow(vectorB, 21);
    var secondMultiplyResult = MultiplySlow(vectorA, 3);
    var finalResult = SumSlow(firstMultiplyResult, secondMultiplyResult);
    return finalResult;
}

private static List<int> CalculateFunctionMultipleThreadSmart(List<int> vectorA,
List<int> vectorB)
{
    var firstMultiplyTask = Task.Run(() => MultiplySlow(vectorB, 21));
    var secondMultiplyTask = Task.Run(() => MultiplySlow(vectorA, 3));

    Task.WaitAll(firstMultiplyTask, secondMultiplyTask);
```

```
    var firstMultiplyResult = firstMultiplyTask.Result;
    var secondMultiplyResult = secondMultiplyTask.Result;
    var finalResult = Task.Run(() => SumSlow(firstMultiplyResult,
secondMultiplyResult)).Result;
    return finalResult;

}
```

Результаты работы программы получились следующие:

Enter q to exit.

Enter size of vectors: 5

vectorA: 1 18 7 9 19

vectorB: 7 11 12 12 19

Single Thread results.

result: 150 285 273 279 456

Single thread, time of execution 5 ms

Multiple Thread results.

Thread 1, results: 49 77 84 84 133

Thread 2, results: 50 95 91 93 152

Thread 3, results: 150 285 273 279 456

result: 150 285 273 279 456

Multiple thread, time of execution 35 ms

Multiple Thread with Delays results.

Thread 1, results: 49 77 84 84 133

Thread 2, results: 50 95 91 93 152

Thread 3, results: 150 285 273 279 456

result: 150 285 273 279 456

Multiple thread with Delays, time of execution 950 ms

Single Thread Smart results.

result: 150 285 273 279 456

Single thread Smart, time of execution 936 ms

Multiple Thread Smart results.

result: 150 285 273 279 456

Multiple thread Smart, time of execution 628 ms

Исходя из полученных данных можно сделать следующие вывод, что без изменения исходной формулы, многопоточный вариант работает медленнее однопоточного. Это связано с тем, что следующий шаг опирается на результаты предыдущего и второй/третий потоки не могут начать выполнение до тех пор, пока не получат результаты от предыдущего потока. Например, мы не можем вычислять операцию сложения, пока не знаем результаты умножения. В качестве решения этой проблемы можно использовать несколько подходов:

1) Разбить программу на потоки другим способом. Так как вычисление операции над вектором (в текущей постановке задачи) выполняется по-координатно, то более правильным и быстрым решением было бы выделять поток под конкретную координату или под подмножество координат, а внутри потока считать не отдельную операцию, а финальное выражение. Это позволило бы потокам не ждать друг друга и выполнять операции над вектором в разы быстрее;
2) Альтернативным решением является корректировка исходной формулы. Не трудно заметить, что при открытии скобок получается сумма двух произведений. Тогда можно в отдельных потоках вычислить каждое из произведений, дождаться их результатов и сложить два произведения. Таким образом работа программы ускоряется примерно на 33%. Именно этот метод и был проверен методами, в названии которых присутствует суффикс «Smart».

Вывод: была написана программа, использующая многопоточный подход к программированию. Программа позволяет сравнить различные подходы к решению задачи и сделать выводы. Программа отлажена и протестирована ручными методами тестирования. Исходный код программы залит на Github и доступен по ссылке: https://github.com/bukSHA1024/RSU_TRPO_Lab3