МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ЛИПЕЦКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Институт компьютерных наук
Кафедра автоматизированных систем управления

Лабораторная работа №3

по операционным системам

Студент АС-21-1 _____ Станиславчук С. М.

(подпись, дата)

Руководитель

Доцент, к.п.н. _____ Кургасов В. В.

(подпись, дата)

Липецк 2024

Содержание

2. Цель работы

Получение практических навыков по работе с подсистемой инициализации и управления службами.

3. Задание
Задание 1. Написать демон, представляющий собой программу для отдыха глаз. Демон будет показывать уведомление о начале отдыха раз в заданный промежуток времени (repetition_period) и уведомление об окончании отдыха через заданный промежуток времени (relax_time). Временные промежутки задаются случайным образом
Вариант 2: Стереокартинки

Задание 2. Корутина
На диске лежат файлы, в них числа в строковом виде, в произвольном порядке, они разделены пробелами. Нужно отсортировать каждый файл и затем слить их в один большой.
То есть выполнить сортировку слиянием.
Реализовать задачу с использованием корутины.
Правила:

- Каждый файл надо сортировать в отдельной корутине. Эта часть задания нацелена на понимание кооперативного планирования задач.
- Файлы имеют кодировку ASCII. То есть это обычный текст, не unicode или что-то такое.
- Необходимо замерить время выполнения операций, используя clock_gettime (CLOCK_MONOTONIC).

Ограничения:

- Глобальные переменные запрещены (кроме уже существующих).
- Для сортировки нельзя использовать встроенные функции типа qsort(), system("sort ...") и т.д.
- Сложность сортировки индивидуальных файлов должна быть < O(N^2) (например, нельзя использовать сортировку пузырьком).
- Суммарное время работы всей программы ограничено. (Для теста можно использовать 6 файлов, каждый с 40к чисел, время не должно занимать больше одной секунды).
- Работа с файлами должна быть
  - либо через числовые файловые дескрипторы и функции open() / read() / write() / close(),
  - либо через FILE* дескрипторы и функции fopen() / fscanf() / fprintf() / fclose(). Нельзя

  использовать std::iostream, std::ostream, std::istream и прочий STL.

- корутины должны переключаться. Делать так называемые 'yield', 'илды'.

Послабления:

- Числа помещаются в 'int'.

- Можно полагать, что все файлы помещаются целиком в память, даже все одновременно.

- Финальный шаг - само слияние сортированных массивов – можно делать прямо в main() снаружи от корутин.

Советы:

- Вы можете найти больше информации о разных новых функциях при помощи консольной
команды 'man'. Наример, 'man read' (или 'man 2 read') напечатает документацию функции
'read()'.
'man strdup' расскажет больше про функцию 'strdup()'. Таким же образом можно искать
другие встроенные функции.

- Шаги, которым можно следовать, если не знаете, с чего начать:

- Реализовать обычную сортировку одного файла. Без корутин, без множества файлов.
Просто прочитать и отсортировать один файл.
Протестируйте этот код.

- Расширьте свой код, чтобы теперь он сортировал много файлов через сортировку слиянием. Без корутин. Проверьте свой код на реальных тестах из задания. Когда он заработает, вы сможете сконцентрироваться на добавлении корутин, и не тратить время на отладку одновременно и корутин, и сортировки, и работы с файлами.

- Встройте в свой код корутины.
Предусмотреть в выводе: суммарное время работы программы, время работы и количество переключений каждой корутины. Учесть, что время работы корутины не включает ее время ожидания. То есть пока она спала во время coro_yield(). Вы должны остановить таймер работы корутины прямо перед coro_yield() и возобновить его сразу после.

4. Программа 1.
Для написания daemon (в моем случае я использую ОС Arch Linux) потребуются следующие компоненты:
1. dbus – шина для возможности получения и отрисовки системных уведомлений
2. python – язык программирования, на котором будет написан демон
3. python-модули:
    1. plyer - для работы с уведомлениями
    2. PIL – для возможности открывать изображения

Код программы:

```
import time
import random
from plyer import notification
from PIL import Image

def show_images(images):
    for image in images:
        img = Image.open(image)
        img.show()

def eye_rest_demon(repetition_period, relax_time, images):
    while True:
        # Случайный выбор времени для начала отдыха
        start_rest_time = random.randint(1, repetition_period)
        # Случайный выбор времени для продолжительности отдыха
        duration_rest = random.randint(1, relax_time)

        # Ожидание начала отдыха
        time.sleep(start_rest_time)

        # Уведомление о начале отдыха
        notification.notify(
            title="Отдых для глаз",
            message="Пора отдохнуть от экрана! Время отдыха: {} мин.".format(duration_rest),
            timeout=10
        )

        # Показ изображений
        show_images(images)

        # Ожидание окончания отдыха
        time.sleep(duration_rest * 60)

        # Закрытие всех открытых изображений
        Image.close()

        # Уведомление об окончании отдыха
        notification.notify(
            title="Время вернуться к работе",
            message="Отдых закончен. Время вернуться к работе!",
            timeout=10
        )

if __name__ == "__main__":
    repetition_period = 60  # Периодичность проверки, в секундах
    relax_time = 2  # Продолжительность отдыха, в минутах
    images = ["image_1.png", "image_2.jpg", "image_3.jpg", "image_4.jpg", "image_5.jpg"]
    eye_rest_demon(repetition_period, relax_time, images)
```

Запустим daemon:

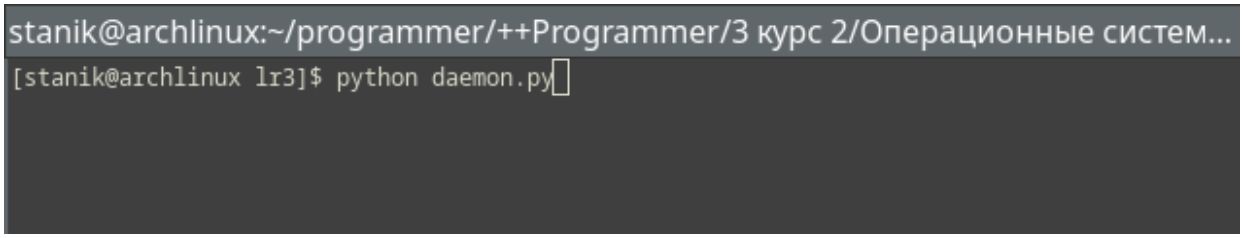


Рисунок 1 - Запуск daemon

Спустя минуту (заданное для тестирования значение переменной) пришло уведомление в правом верхнем углу экрана (по умолчанию):

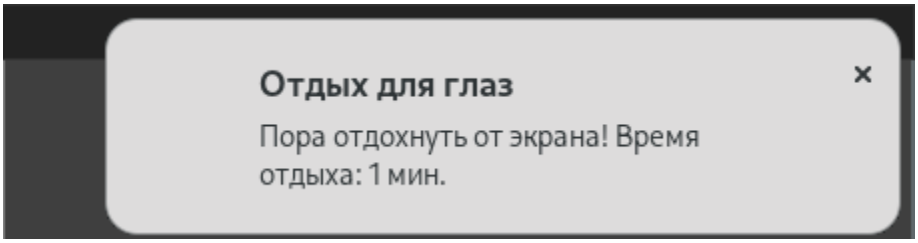


Рисунок 2 - Вызов и отрисовка уведомления, в данном случае с одноминутным отдыхом

Вывод изображений (отдых для глаз) в отдельном окне, в моем случае – Google Chrome

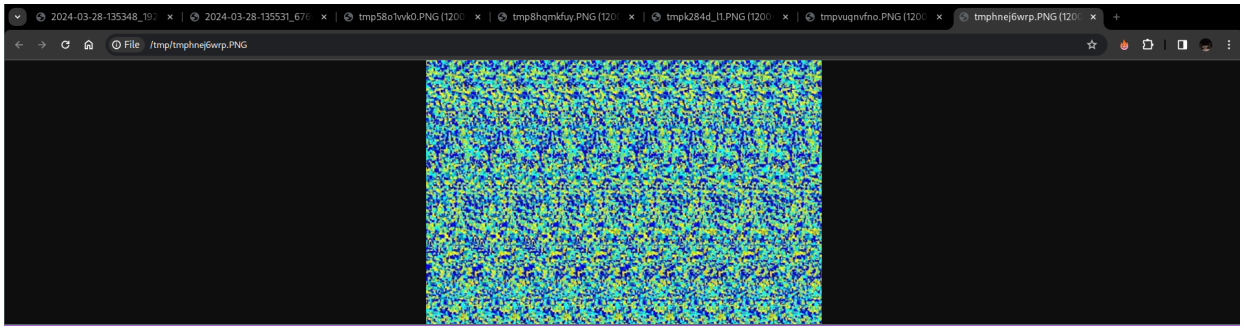


Рисунок 3 – Одно из открывшихся изображений (Стереокартинка)

Демон работает в бесконечном цикле и будет напоминать пользователю об отдыхе каждые rand(n, m) минут.

Что делает этот обычный скрипт демоном? - Лишь формат процесса. Демон – это точно такая же программа, она лишь выполняется в фоновом процессе. То есть, теперь, чтобы сделать скрипт демоном нужно либо:

запустить данную программу, используя ‘&’ или же прописать нужные параметры в конфиг файле systemd. Допустим, мы хотим чтобы этот демон тихо работал и включался каждый раз при входе в систему. Тогда делаем так: Перейдим в каталог `/etc/systemd/system/` и создаем файл расширением .service, например, `mydaemon.service` со следующими параметрами:

```
[Unit]
Description=My Daemon Service

[Service]
ExecStart=/usr/bin/mydaemon
Restart=always

[Install]
WantedBy=multi-user.target
```

Теперь наш скрипт стал демоном, который работает всегда в фоне и его совсем не нужно запускать вручную, так как он сам инициализируется и запускается в фоне при старте системы.

Программа 2.

Реализую данный алгоритм в Unity, C#. В чем суть? Надо написать пару скриптов: один будет генерировать 6 текстовых файлов, которые будут хранить 240 000 / 6 → по 40 000 случайных чисел; второй скрипт будет создавать корутины для каждого процесса сортировки файлов слиянием; третий же скрипт будет хранить используемый во втором метод сортировки.

Скрипт, генерирующий файлы:

```
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;

public class FileGenerator : MonoBehaviour
{
    private const uint numFiles = 6;
    private const int numNumbers = 40000;
    private const int maxNumberRoofRange = numNumbers * 100;

    private void Awake()
    {
        GenerateFilesRand();
    }


    private void GenerateFilesRand()
```

```
    {
        for (int i = 0; i < numFiles; i++)
        {
            string fileName = Path.Combine(Application.persistentDataPath, $"file{i+1}.txt");
            GenerateFileRand(fileName, numNumbers);
        }
    }

    private void GenerateFileRand(string fileName, int numNumbers)
    {
        using (StreamWriter writer = new StreamWriter(fileName))
        {
            for (int i = 0; i < numNumbers; i++)
            {
                writer.Write(UnityEngine.Random.Range(0, maxNumberRoofRange));
                if (i < numNumbers - 1)
                    writer.Write(" ");
            }
        }

        Debug.Log("File with random int's generated: " + fileName);
    }


    public void SaveOutputFile(int[] arr)
    {
        // Путь к файлу, куда будем сохранять массив
        string filePath = Path.Combine(Application.persistentDataPath, "output.txt");

        // Создаем новый файл или перезаписываем существующий
        using (StreamWriter writer = new StreamWriter(filePath))
        {
            // Записываем каждый элемент массива в файл
            foreach (int num in arr)
            {
                writer.WriteLine(num.ToString());
            }
        }

        Debug.Log("Array saved to file: " + filePath);
    }

}
```

Скрипт, создающий сортирующие корутины, с таймерами:

```
using UnityEngine;
using System.Collections;
using System.IO;
using System;
using System.Diagnostics;
using System.Collections.Generic;
using static SortingAlgorithm;


public class ParallelFileSorting : MonoBehaviour
{
    private List<int[]> sortedArrays = new List<int[]>();
    private List<int[]> mergedArrays = new List<int[]>();

    private double totalSortingTime = 0.0;
    private double totalIndSortTIme = 0.0;
    private int finishedCoroutines = 0;
    private readonly object lockObject = new object();

    [SerializeField] private FileGenerator fileGen;


    private void Awake()
    {

    }
```

```
    private void Start()
    {
        UnityEngine.Debug.Log("<b>Parallel Sorting</b>");
        StartCoroutine(SortFiles());
    }

    private IEnumerator SortFiles()
    {
        string[] files = Directory.GetFiles(Application.persistentDataPath, "*.txt");
        List<Coroutine> coroutines = new List<Coroutine>();

        Stopwatch stopwatch = Stopwatch.StartNew();

        foreach (string file in files)
        {
            coroutines.Add(StartCoroutine(SortFileCoroutine(file)));
        }
        stopwatch.Stop();
        TimeSpan timeSpan = stopwatch.Elapsed;

        UnityEngine.Debug.Log($"Sorting files (with self-Coroutine for each file) took
<b>{timeSpan.TotalMilliseconds}</b> ms");

        // Ждем завершения всех корутин
        foreach (Coroutine coroutine in coroutines)
        {
            yield return coroutine;
        }

        // Выполняем слияние отсортированных массивов
        mergedArrays.Add(sortedArrays[0]);
        for (int i = 1; i < sortedArrays.Count; i++)
        {
            int[] merged = Merge(mergedArrays[i-1], sortedArrays[i]);
            mergedArrays.Add(merged);
        }

        // Выводим результат слияния
        UnityEngine.Debug.Log("Merged Array: " + string.Join(", ", mergedArrays[mergedArrays.Count - 1]));
        fileGen.SaveOutputFile(mergedArrays[mergedArrays.Count - 1]);
        UnityEngine.Debug.Log("Merged Array written to output.txt");

        UnityEngine.Debug.Log("totalSortingTime = " + totalSortingTime);
        yield return null;
    }

    private IEnumerator SortFileCoroutine(string filePath)
    {
        int[] array = ParseAndSort(filePath);
        sortedArrays.Add(array);

        // Если все корутины завершились, можно продолжить
        while (finishedCoroutines == sortedArrays.Capacity)
        {
            yield break;
        }
    }

    private int[] ParseAndSort(string filePath)
    {
        // Загружаем данные из файла
        string data = File.ReadAllText(filePath);
        string[] stringNumbers = data.Split(' ');

        // Преобразуем данные в массив чисел
        List<int> numbersList = new List<int>();
        foreach (string str in stringNumbers)
        {
            if (int.TryParse(str, out int number))
            {
                numbersList.Add(number);
            }
            else
```

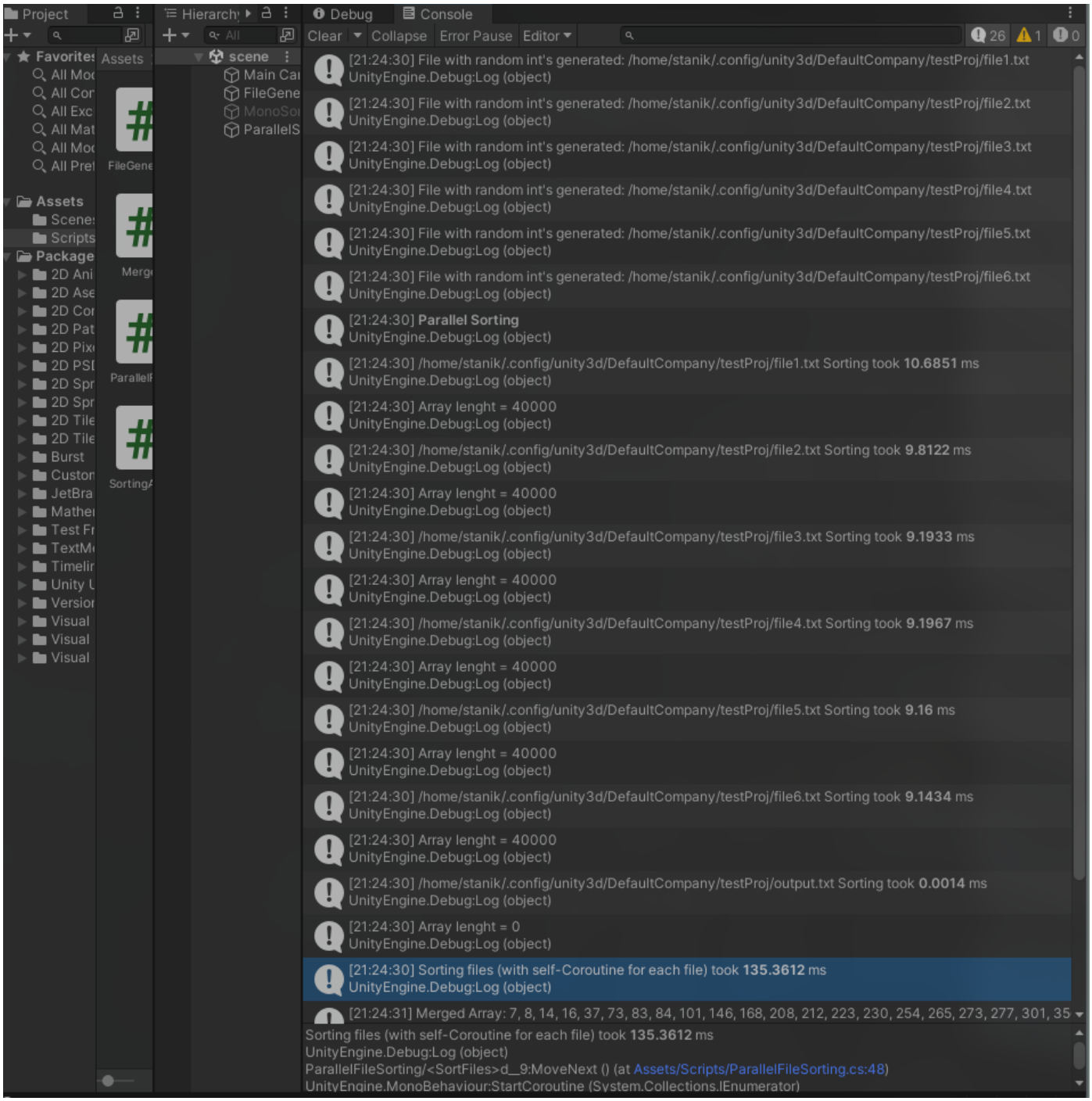
```
            {
                UnityEngine.Debug.LogWarning($"Unable to parse '{str}' as integer.");
            }
        }

        // Преобразуем список чисел в массив
        int[] array = numbersList.ToArray();

        // Засекаем время начала сортировки
        Stopwatch stopwatch = Stopwatch.StartNew();
        // Сортируем массив чисел
        SortingAlgorithm.MergeSort(array);
        // Засекаем время окончания сортировки
        stopwatch.Stop();

        TimeSpan timeSpan = stopwatch.Elapsed;
        UnityEngine.Debug.Log($"{filePath} Sorting took <b>{timeSpan.TotalMilliseconds}</b> ms");

        UnityEngine.Debug.Log("Array lenght = " + numbersList.Count);
        totalSortingTime += timeSpan.TotalMilliseconds;
        return array;
    }
}
```

Скрипт #3 содержит алгоритм сортировки:

```
public static class SortingAlgorithm
{
    public static void MergeSort(int[] array)
    {
        if (array == null || array.Length <= 1)
            return;

        MergeSortRecursive(array, 0, array.Length - 1);
    }

    public static int[] Merge(int[] arr1, int[] arr2)
    {
        int[] merged = new int[arr1.Length + arr2.Length];
        int i = 0, j = 0, k = 0;

        while (i < arr1.Length && j < arr2.Length)
        {
            if (arr1[i] < arr2[j])
            {
                merged[k] = arr1[i];
                i++;
            }
            else
            {
                merged[k] = arr2[j];
                j++;
            }
            k++;
        }

        while (i < arr1.Length)
        {
            merged[k] = arr1[i];
            i++;
            k++;
        }

        while (j < arr2.Length)
        {
            merged[k] = arr2[j];
            j++;
            k++;
        }
```

```
        return merged;
    }


    private static void MergeSortRecursive(int[] array, int left, int right)
    {
        if (left < right)
        {
            int mid = (left + right) / 2;

            MergeSortRecursive(array, left, mid);
            MergeSortRecursive(array, mid + 1, right);

            Merge(array, left, mid, right);
        }
    }


    private static void Merge(int[] array, int left, int mid, int right)
    {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int[] leftArray = new int[n1];
        int[] rightArray = new int[n2];

        for (int i = 0; i < n1; ++i)
            leftArray[i] = array[left + i];
        for (int j = 0; j < n2; ++j)
            rightArray[j] = array[mid + 1 + j];

        int p = 0, q = 0, k = left;
        while (p < n1 && q < n2)
        {
            if (leftArray[p] <= rightArray[q])
            {
                array[k] = leftArray[p];
                p++;
            }
            else
            {
                array[k] = rightArray[q];
                q++;
            }
            k++;
        }

        while (p < n1)
        {
            array[k] = leftArray[p];
            p++;
            k++;
        }

        while (q < n2)
        {
            array[k] = rightArray[q];
            q++;
            k++;
        }
    }
}
```

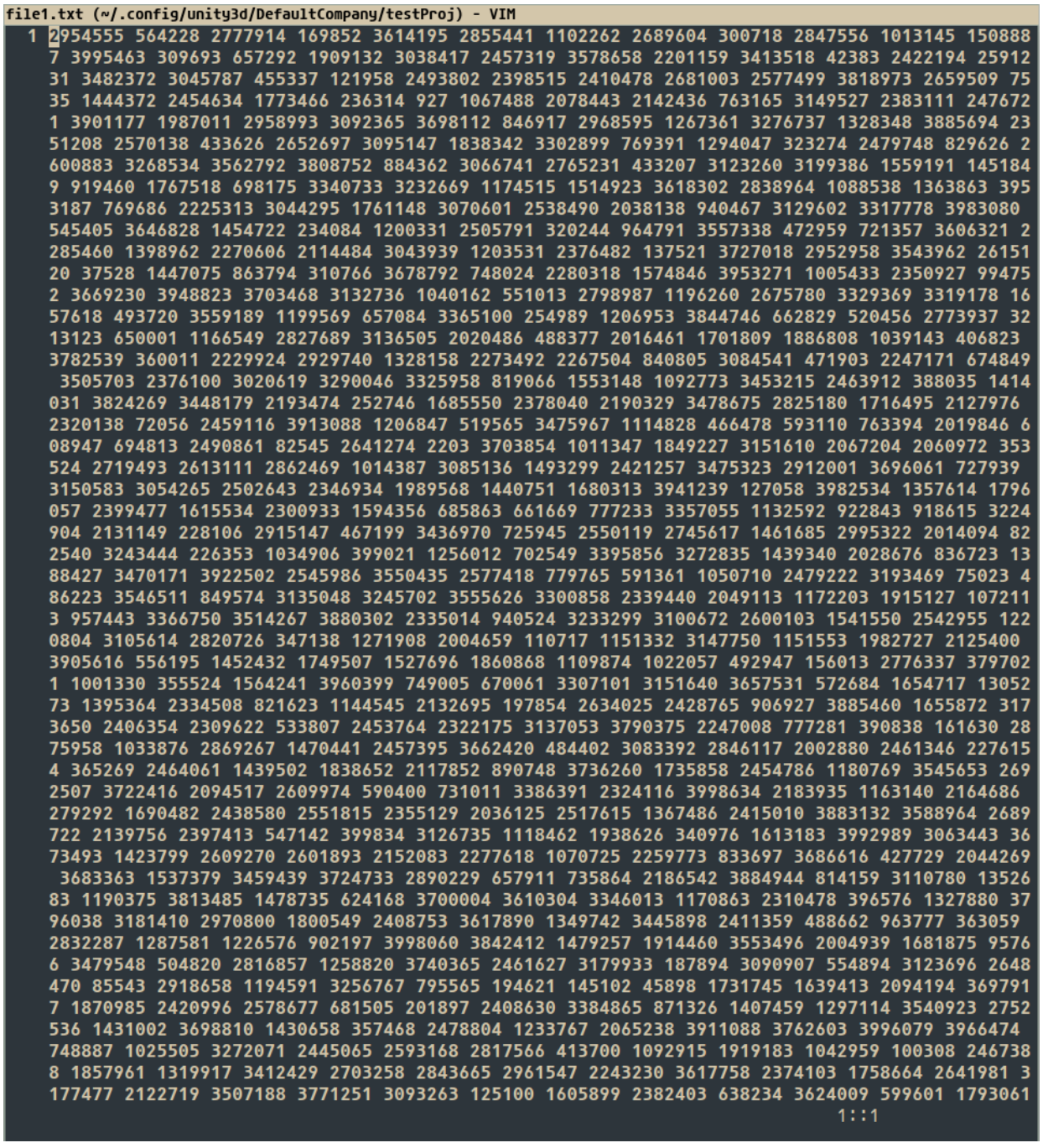
Теперь протестируем программу.
Создадим файлы с названиями file_n.txt и output.txt
Запускаем программу и смотрим лог.



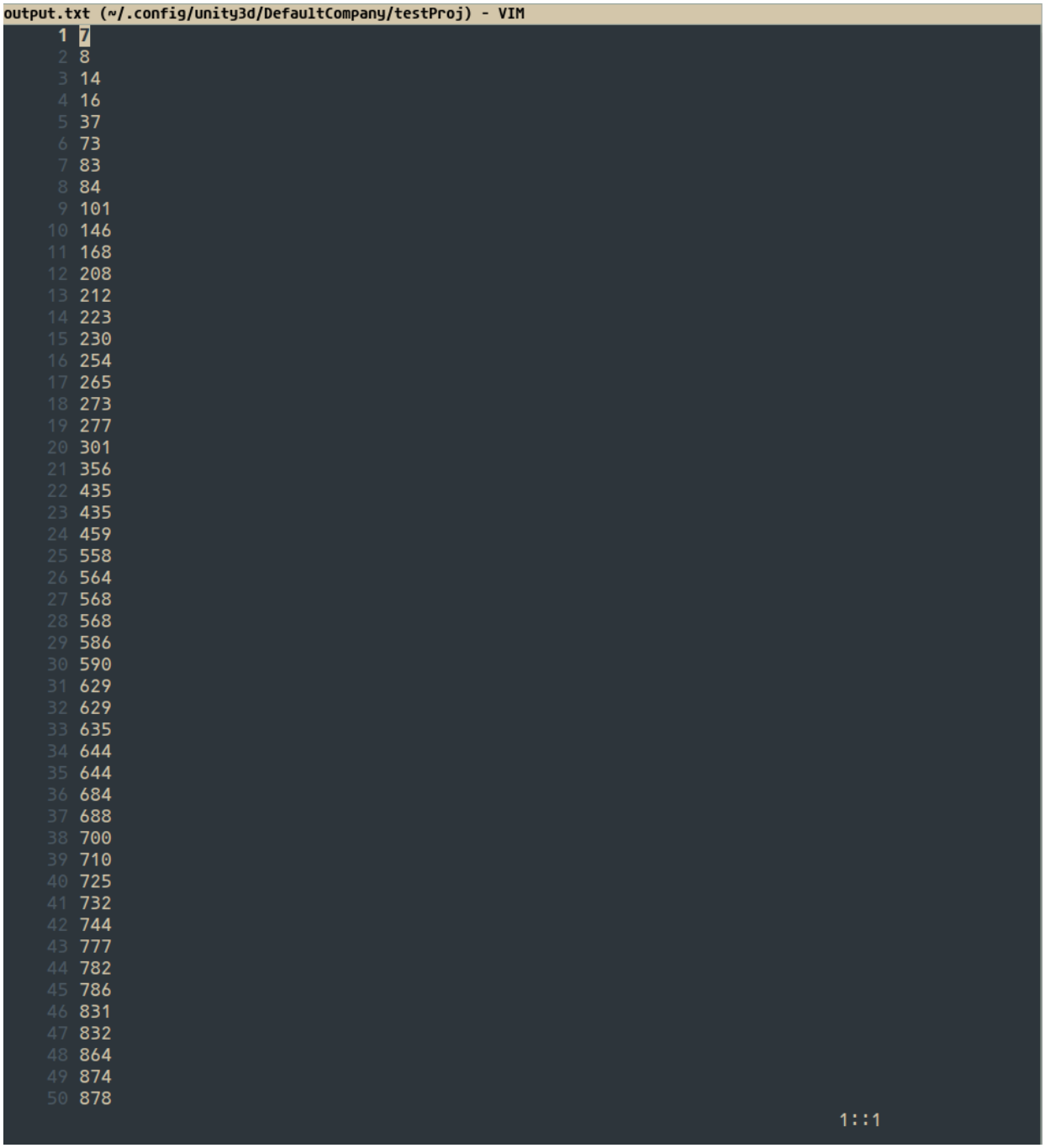
Как видно из логов, на сортировку каждого файла, а также их слияние в один большой файл, размером 240 000 целочисленных значений, без учета времени генерации файлов с числами, заняло около 135 миллисекунд, что

является более чем успешным результатом. Таким образом, используя корутины, я задействовал сразу несколько потоков своего процессора, чтобы отсортировать конечный файл за максимально малое время.

Пример сгенерированного файла file_1.txt:



Полученный в результате сортировки файл:



Вывод: в ходе выполненной работы создал демон-процесс и написал программу, параллельно сортирующую несколько файлов, после чего объединяет их в один.