МИНИСТЕРСТВО ТРАНСПОРТА РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА (МИИТ)»

Кафедра ЦТУТП

**ОТЧЁТ**

**ПО КУРСОВОЙ РАБОТЕ**   
по дисциплине «Операционные системы и системное программирование»

Тема: «ОРГАНИЗАЦИЯ МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЙ В ОС WINDOWS»

Выполнил: Русов В. Д.

Группа: УИС-311

Преподаватель: доц. Варфоломеев В. А.

­

Москва 2024 г.

**СОДЕРЖАНИЕ**

[ВАРИАНТ ЗАДАНИЯ 3](#_Toc176644761)

[БЛОК-СХЕМА АЛГОРИТМА 4](#_Toc176644762)

[ТЕКСТ ПРОГРАММ 6](#_Toc176644763)

[Программа A 6](#_Toc176644764)

[Программа B 13](#_Toc176644765)

[Программа C 21](#_Toc176644766)

[РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММ 30](#_Toc176644767)

[СКРИНШОТЫ ОКНА УТИЛИТЫ PROCESS EXPLORER 33](#_Toc176644768)

[ОПИСАНИЕ И РЕЗУЛЬТАТЫ ПРОВЕДЕННОГО ЭКСПЕРИМЕНТА 35](#_Toc176644769)

[ДИАГРАММЫ ТРАСС ПОТОКОВ 40](#_Toc176644770)

[ВЫВОДЫ ПО РЕЗУЛЬТАТАМ ПРОВЕДЕННГО ЭКСПЕРИМЕНТА 42](#_Toc176644771)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 44](#_Toc176644772)

# ВАРИАНТ ЗАДАНИЯ

Разработать три консольные программы, выполняющие обработку заданного множества текстовых файлов в соответствии с индивидуальным заданием (см. табл.1):

A – однопоточная программа с последовательной (циклической) обработкой файлов;

B – многопоточная программа с параллельной обработкой файлов;

C – многопоточная программа с параллельной обработкой файлов и повышенным приоритетом потоков.

В каждой программе производить измерение общего времени обработки всех файлов и время, затраченное на обработку каждого файла в отдельности. Программы должны содержать шапку в виде комментария с указанием фамилии студента и номера группы, номера варианта и назначения программы.

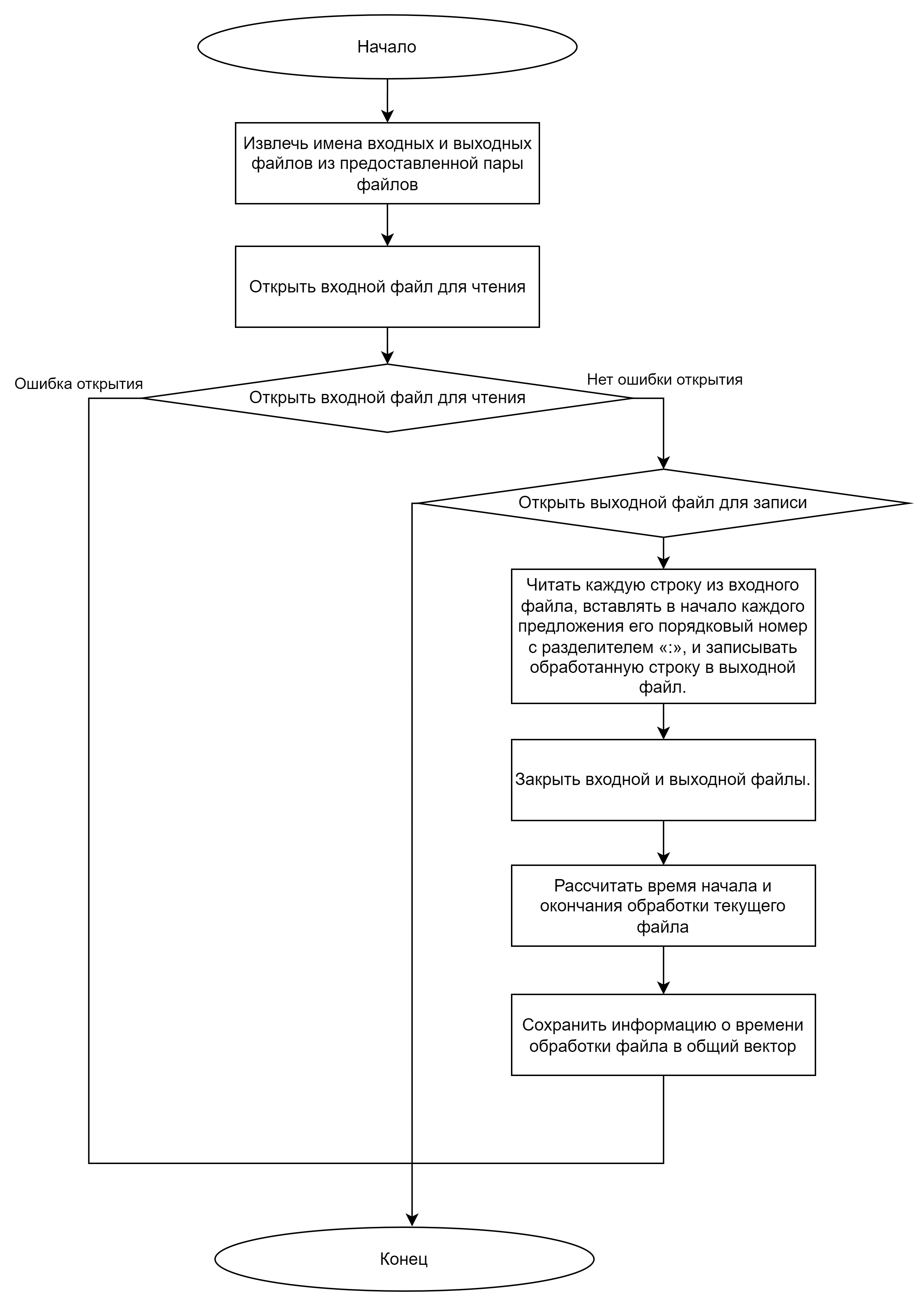
Вариант: 14

Таблица 1 – Вариант задания

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | № | Выполняемые действия | | 14 | Вставить в начало каждого предложения его порядковый номер с разделителем «:» | |  |
|  |  |

# БЛОК-СХЕМА АЛГОРИТМА





# ТЕКСТ ПРОГРАММ

## **Программа A**

#include <iostream>

#include <fstream>

#include <sstream>

#include <string>

#include <unordered\_set>

#include <vector>

#include <algorithm>

#include <ctime>

#include "util.h"

// Structure for storing file processing time

struct FileProcessingTime {

    std::string filename;

    clock\_t start\_time;

    clock\_t end\_time;

};

std::vector<FileProcessingTime> fileProcessingTimes;

std::unordered\_set<std::string> fileNamesSet;

clock\_t startTime;

clock\_t endTime;

// Initialization of directories

std::string inputDirectory = "abstract\_path\\in\\";

std::string outputDirectory = "abstract\_path\\out\\";

// Global counter for sentences

size\_t globalSentenceCounter = 0;

void processFile(const std::string &inputFilename, const std::string &outputFilename) {

    FileProcessingTime processingTime;

    clock\_t startFileTime, endFileTime;

    startFileTime = clock();

    processingTime.filename = inputFilename;

    std::ifstream inFile(inputFilename);

    if (!inFile.is\_open()) {

        std::cerr << "Error: Couldn't open input file '" << inputFilename << "'." << std::endl;

        return;

    }

    std::ofstream outFile(outputFilename);

    if (!outFile.is\_open()) {

        std::cerr << "Error: Couldn't open output file '" << outputFilename << "'." << std::endl;

        inFile.close();

        return;

    }

    std::string line;

    // Process each line of the input file

    while (std::getline(inFile, line)) {

        // Remove leading and trailing spaces

        line.erase(0, line.find\_first\_not\_of(" \t"));

        line.erase(line.find\_last\_not\_of(" \t") + 1);

        // Split the line into sentences

        std::string delimiters = ".!?"; // Sentence delimiters

        std::vector<std::string> sentences;

        size\_t pos = 0;

        // Extract sentences

        while ((pos = line.find\_first\_of(delimiters)) != std::string::npos) {

            std::string sentence = line.substr(0, pos + 1); // Include the delimiter

            sentences.push\_back(sentence);

            line.erase(0, pos + 1); // Remove the extracted sentence from the line

        }

        // If there's remaining text, add it as the last sentence

        if (!line.empty()) {

            sentences.push\_back(line);

        }

        // Numbering and writing sentences to the output file

        for (const auto &sentence : sentences) {

            std::string trimmedSentence = sentence;

            trimmedSentence.erase(0, trimmedSentence.find\_first\_not\_of(" \t")); // Remove leading spaces

            trimmedSentence.erase(trimmedSentence.find\_last\_not\_of(" \t") + 1); // Remove trailing spaces

            if (!trimmedSentence.empty()) {

                outFile << (++globalSentenceCounter) << ": " << trimmedSentence << std::endl; // Use global counter

            }

        }

    }

    inFile.close();

    outFile.close();

    endFileTime = clock();

    processingTime.start\_time = startFileTime;

    processingTime.end\_time = endFileTime;

    fileProcessingTimes.push\_back(processingTime);

    std::cout << "Lines sorted and saved to '" << outputFilename << "'." << std::endl;

}

// Function to write processing time to a text file

void writeToTextFile(const std::string &filename) {

    std::string outputFilename = outputDirectory + filename;

    std::ofstream outFile(outputFilename);

    if (outFile.is\_open()) {

        outFile << "Program Start Time: " << static\_cast<double>(startTime) / CLOCKS\_PER\_SEC << " seconds\n";

        outFile << "Program End Time: " << static\_cast<double>(endTime) / CLOCKS\_PER\_SEC << " seconds\n";

        outFile << "Program Elapsed Time: " << static\_cast<double>(endTime - startTime) / CLOCKS\_PER\_SEC << " seconds\n";

        outFile << "\nFile Processing Times:\n";

        for (const auto &processingTime : fileProcessingTimes) {

            outFile << "File: " << processingTime.filename

                    << ", Start Time: " << static\_cast<double>(processingTime.start\_time) / CLOCKS\_PER\_SEC << " seconds"

                    << ", End Time: " << static\_cast<double>(processingTime.end\_time) / CLOCKS\_PER\_SEC << " seconds"

                    << ", Elapsed Time: " << static\_cast<double>(processingTime.end\_time - processingTime.start\_time) / CLOCKS\_PER\_SEC << " seconds\n";

        }

        outFile.close();

        std::cout << "Data written to text file '" << outputFilename << "'." << std::endl;

    } else {

        std::cerr << "Error writing to file '" << outputFilename << "'." << std::endl;

    }

}

int main(int argc, char \*argv[]) {

    startTime = clock();

    if (argc < 3) {

        std::cerr << "Usage: " << argv[0] << " <path\_type> <filename1> [<filename2> ... <filenameN>]" << std::endl;

        std::cerr << "<path\_type> should be either '--removable' or '--harddisk'" << std::endl;

        return 1;

    }

    std::string pathType = argv[1];

    if (pathType == "--removable") {

        inputDirectory = "E:\\files\\in\\";

        outputDirectory = "E:\\files\\out\\";

    } else if (pathType == "--harddisk") {

        std::wstring executableDir = getDirectoryOfExecutable();

        inputDirectory = std::string(executableDir.begin(), executableDir.end()) + "\\files\\in\\";

        outputDirectory = std::string(executableDir.begin(), executableDir.end()) + "\\files\\out\\";

    } else {

        std::cerr << "Error: Unknown path type '" << pathType << "'." << std::endl;

        std::cerr << "<path\_type> should be either '--removable' or '--harddisk'" << std::endl;

        return 1;

    }

    for (int i = 2; i < argc; ++i) {

        std::string inputFilename = inputDirectory + argv[i];

        if (fileNamesSet.find(inputFilename) != fileNamesSet.end()) {

            std::cout << "File '" << inputFilename << "' reoccurs. Skipping." << std::endl;

            continue;

        }

        fileNamesSet.insert(inputFilename);

        std::string outputFilename = outputDirectory + "out\_" + argv[i];

        processFile(inputFilename, outputFilename);

    }

    endTime = clock();

    writeToTextFile("file\_processing\_times\_a.txt");

    return 0;

}

## **Программа B**

#include <windows.h>

#include <iostream>

#include <fstream>

#include <sstream>

#include <string>

#include <unordered\_set>

#include <vector>

#include <algorithm>

#include <ctime>

#include "util.h"

// Struct to store file processing time details

struct FileProcessingTime {

    std::string filename;

    clock\_t start\_time;

    clock\_t end\_time;

};

std::vector<FileProcessingTime> fileProcessingTimes;

std::unordered\_set<std::string> fileNamesSet;

clock\_t startTime;

clock\_t endTime;

CRITICAL\_SECTION cs; // Critical section for thread safety

// Initializing default directories

std::string inputDirectory = "abstract\_path\\in\\";

std::string outputDirectory = "abstract\_path\\out\\";

// Function to trim leading and trailing whitespace

std::string trimWhitespace(const std::string &str) {

    auto start = str.find\_first\_not\_of(" \t");

    auto end = str.find\_last\_not\_of(" \t");

    return (start == std::string::npos || end == std::string::npos) ? "" : str.substr(start, end - start + 1);

}

// Thread function to process a file and write sentences to an output file

DWORD WINAPI processFileThread(LPVOID lpParam) {

    auto \*filePair = reinterpret\_cast<std::pair<std::string, std::string> \*>(lpParam);

    std::string inputFilename = filePair->first;

    std::string outputFilename = filePair->second;

    delete filePair; // Free memory

    FileProcessingTime processingTime;

    clock\_t startFileTime, endFileTime;

    startFileTime = clock();

    processingTime.filename = inputFilename;

    std::ifstream inFile(inputFilename);

    if (!inFile.is\_open()) {

        std::cerr << "Error: Couldn't open input file '" << inputFilename << "'." << std::endl;

        return 1;

    }

    std::ofstream outFile(outputFilename);

    if (!outFile.is\_open()) {

        std::cerr << "Error: Couldn't open output file '" << outputFilename << "'." << std::endl;

        inFile.close();

        return 1;

    }

    std::string line;

    size\_t sentenceCounter = 0; // Local counter for sentences

    // Process each line of the input file

    while (std::getline(inFile, line)) {

        line = trimWhitespace(line); // Trim spaces

        std::string delimiters = ".!?"; // Sentence delimiters

        size\_t pos = 0;

        // Split the line into sentences

        while ((pos = line.find\_first\_of(delimiters)) != std::string::npos) {

            std::string sentence = line.substr(0, pos + 1); // Include delimiter

            outFile << (++sentenceCounter) << ": " << trimWhitespace(sentence) << std::endl; // Write sentence

            line.erase(0, pos + 1); // Remove the extracted sentence

        }

        // Write any remaining text as the last sentence

        if (!line.empty()) {

            outFile << (++sentenceCounter) << ": " << trimWhitespace(line) << std::endl;

        }

    }

    inFile.close();

    outFile.close();

    endFileTime = clock();

    processingTime.start\_time = startFileTime;

    processingTime.end\_time = endFileTime;

    EnterCriticalSection(&cs); // Enter critical section for thread safety

    fileProcessingTimes.push\_back(processingTime); // Save processing time

    LeaveCriticalSection(&cs); // Leave critical section

    std::cout << "Lines processed and saved to '" << outputFilename << "'." << std::endl;

    return 0;

}

// Function to write processing times to a text file

void writeToTextFile(const std::string &filename) {

    std::string outputFilename = outputDirectory + filename;

    std::ofstream outFile(outputFilename);

    if (outFile.is\_open()) {

        outFile << "Program Start Time: " << static\_cast<double>(startTime) / CLOCKS\_PER\_SEC << " seconds\n";

        outFile << "Program End Time: " << static\_cast<double>(endTime) / CLOCKS\_PER\_SEC << " seconds\n";

        outFile << "Program Elapsed Time: " << static\_cast<double>(endTime - startTime) / CLOCKS\_PER\_SEC << " seconds\n";

        outFile << "\nFile Processing Times:\n";

        for (const auto &processingTime : fileProcessingTimes) {

            outFile << "File: " << processingTime.filename

                    << ", Start Time: " << static\_cast<double>(processingTime.start\_time) / CLOCKS\_PER\_SEC << " seconds"

                    << ", End Time: " << static\_cast<double>(processingTime.end\_time) / CLOCKS\_PER\_SEC << " seconds"

                    << ", Elapsed Time: " << static\_cast<double>(processingTime.end\_time - processingTime.start\_time) / CLOCKS\_PER\_SEC << " seconds\n";

        }

        outFile.close();

        std::cout << "Data written to text file '" << outputFilename << "'." << std::endl;

    } else {

        std::cerr << "Error writing to file '" << outputFilename << "'." << std::endl;

    }

}

int main(int argc, char \*argv[]) {

    startTime = clock();

    InitializeCriticalSection(&cs); // Initialize critical section

    if (argc < 3) {

        std::cerr << "Usage: " << argv[0] << " <path\_type> <filename1> [<filename2> ... <filenameN>]" << std::endl;

        std::cerr << "<path\_type> should be either '--removable' or '--harddisk'" << std::endl;

        return 1;

    }

    // Determine input and output directories based on path type

    std::string pathType = argv[1];

    if (pathType == "--removable") {

        inputDirectory = "E:\\files\\in\\";

        outputDirectory = "E:\\files\\out\\";

    } else if (pathType == "--harddisk") {

        std::wstring executableDir = getDirectoryOfExecutable();

        inputDirectory = std::string(executableDir.begin(), executableDir.end()) + "\\files\\in\\";

        outputDirectory = std::string(executableDir.begin(), executableDir.end()) + "\\files\\out\\";

    } else {

        std::cerr << "Error: Unknown path type '" << pathType << "'." << std::endl;

        std::cerr << "<path\_type> should be either '--removable' or '--harddisk'" << std::endl;

        return 1;

    }

    std::vector<HANDLE> threadHandles;

    // Create threads for each input file

    for (int i = 2; i < argc; ++i) {

        std::string inputFilename = inputDirectory + argv[i];

        if (fileNamesSet.find(inputFilename) != fileNamesSet.end()) {

            std::cout << "File '" << inputFilename << "' reoccurs. Skipping." << std::endl;

            continue;

        }

        fileNamesSet.insert(inputFilename);

        std::string outputFilename = outputDirectory + "out\_" + argv[i];

        auto \*filePair = new std::pair<std::string, std::string>(inputFilename, outputFilename);

        HANDLE hThread = CreateThread(NULL, 0, processFileThread, filePair, 0, NULL);

        if (hThread == NULL) {

            std::cerr << "Error creating thread for file '" << inputFilename << "'." << std::endl;

            delete filePair; // Clean up memory in case of thread creation failure

        } else {

            threadHandles.push\_back(hThread); // Store thread handle

        }

    }

    // Wait for all threads to complete

    WaitForMultipleObjects(threadHandles.size(), threadHandles.data(), TRUE, INFINITE);

    // Close thread handles

    for (auto &hThread : threadHandles) {

        CloseHandle(hThread);

    }

    endTime = clock();

    writeToTextFile("file\_processing\_times\_b.txt");

    DeleteCriticalSection(&cs); // Clean up critical section

    return 0;

}

## **Программа C**

#include <windows.h>

#include <iostream>

#include <fstream>

#include <sstream>

#include <string>

#include <unordered\_set>

#include <vector>

#include <algorithm>

#include <ctime>

#include "util.h"

// Struct to store file processing time details

struct FileProcessingTime {

    std::string filename;

    clock\_t start\_time;

    clock\_t end\_time;

};

std::vector<FileProcessingTime> fileProcessingTimes;

std::unordered\_set<std::string> fileNamesSet;

clock\_t startTime;

clock\_t endTime;

CRITICAL\_SECTION cs; // Critical section for thread safety

// Initializing default directories

std::string inputDirectory = "abstract\_path\\in\\";

std::string outputDirectory = "abstract\_path\\out\\";

// Function to trim leading and trailing whitespace

std::string trimWhitespace(const std::string &str) {

    auto start = str.find\_first\_not\_of(" \t");

    auto end = str.find\_last\_not\_of(" \t");

    return (start == std::string::npos || end == std::string::npos) ? "" : str.substr(start, end - start + 1);

}

// Thread function to process a file and write sentences to an output file

DWORD WINAPI processFileThread(LPVOID lpParam) {

    auto \*filePair = reinterpret\_cast<std::pair<std::string, std::string> \*>(lpParam);

    std::string inputFilename = filePair->first;

    std::string outputFilename = filePair->second;

    delete filePair; // Free memory

    FileProcessingTime processingTime;

    clock\_t startFileTime, endFileTime;

    startFileTime = clock();

    processingTime.filename = inputFilename;

    std::ifstream inFile(inputFilename);

    if (!inFile.is\_open()) {

        std::cerr << "Error: Couldn't open input file '" << inputFilename << "'." << std::endl;

        return 1;

    }

    std::ofstream outFile(outputFilename);

    if (!outFile.is\_open()) {

        std::cerr << "Error: Couldn't open output file '" << outputFilename << "'." << std::endl;

        inFile.close();

        return 1;

    }

    std::string line;

    size\_t sentenceCounter = 0; // Local counter for sentences

    // Process each line of the input file

    while (std::getline(inFile, line)) {

        line = trimWhitespace(line); // Trim spaces

        std::string delimiters = ".!?"; // Sentence delimiters

        size\_t pos = 0;

        // Split the line into sentences

        while ((pos = line.find\_first\_of(delimiters)) != std::string::npos) {

            std::string sentence = line.substr(0, pos + 1); // Include delimiter

            outFile << (++sentenceCounter) << ": " << trimWhitespace(sentence) << std::endl; // Write sentence

            line.erase(0, pos + 1); // Remove the extracted sentence

        }

        // Write any remaining text as the last sentence

        if (!line.empty()) {

            outFile << (++sentenceCounter) << ": " << trimWhitespace(line) << std::endl;

        }

    }

    inFile.close();

    outFile.close();

    endFileTime = clock();

    processingTime.start\_time = startFileTime;

    processingTime.end\_time = endFileTime;

    EnterCriticalSection(&cs); // Enter critical section for thread safety

    fileProcessingTimes.push\_back(processingTime); // Save processing time

    LeaveCriticalSection(&cs); // Leave critical section

    std::cout << "Lines processed and saved to '" << outputFilename << "'." << std::endl;

    return 0;

}

// Function to write processing times to a text file

void writeToTextFile(const std::string &filename) {

    std::string outputFilename = outputDirectory + filename;

    std::ofstream outFile(outputFilename);

    if (outFile.is\_open()) {

        outFile << "Program Start Time: " << static\_cast<double>(startTime) / CLOCKS\_PER\_SEC << " seconds\n";

        outFile << "Program End Time: " << static\_cast<double>(endTime) / CLOCKS\_PER\_SEC << " seconds\n";

        outFile << "Program Elapsed Time: " << static\_cast<double>(endTime - startTime) / CLOCKS\_PER\_SEC << " seconds\n";

        outFile << "\nFile Processing Times:\n";

        for (const auto &processingTime : fileProcessingTimes) {

            outFile << "File: " << processingTime.filename

                    << ", Start Time: " << static\_cast<double>(processingTime.start\_time) / CLOCKS\_PER\_SEC << " seconds"

                    << ", End Time: " << static\_cast<double>(processingTime.end\_time) / CLOCKS\_PER\_SEC << " seconds"

                    << ", Elapsed Time: " << static\_cast<double>(processingTime.end\_time - processingTime.start\_time) / CLOCKS\_PER\_SEC << " seconds\n";

        }

        outFile.close();

        std::cout << "Data written to text file '" << outputFilename << "'." << std::endl;

    } else {

        std::cerr << "Error writing to file '" << outputFilename << "'." << std::endl;

    }

}

int main(int argc, char \*argv[]) {

    startTime = clock();

    InitializeCriticalSection(&cs); // Initialize critical section

    if (argc < 3) {

        std::cerr << "Usage: " << argv[0] << " <path\_type> <filename1> [<filename2> ... <filenameN>]" << std::endl;

        std::cerr << "<path\_type> should be either '--removable' or '--harddisk'" << std::endl;

        return 1;

    }

    // Determine input and output directories based on path type

    std::string pathType = argv[1];

    if (pathType == "--removable") {

        inputDirectory = "E:\\files\\in\\";

        outputDirectory = "E:\\files\\out\\";

    } else if (pathType == "--harddisk") {

        std::wstring executableDir = getDirectoryOfExecutable();

        inputDirectory = std::string(executableDir.begin(), executableDir.end()) + "\\files\\in\\";

        outputDirectory = std::string(executableDir.begin(), executableDir.end()) + "\\files\\out\\";

    } else {

        std::cerr << "Error: Unknown path type '" << pathType << "'." << std::endl;

        std::cerr << "<path\_type> should be either '--removable' or '--harddisk'" << std::endl;

        return 1;

    }

    std::vector<HANDLE> threadHandles;

    // Create threads for each input file

    for (int i = 2; i < argc; ++i) {

        std::string inputFilename = inputDirectory + argv[i];

        if (fileNamesSet.find(inputFilename) != fileNamesSet.end()) {

            std::cout << "File '" << inputFilename << "' reoccurs. Skipping." << std::endl;

            continue;

        }

        fileNamesSet.insert(inputFilename);

        std::string outputFilename = outputDirectory + "out\_" + argv[i];

        auto \*filePair = new std::pair<std::string, std::string>(inputFilename, outputFilename);

        HANDLE hThread = CreateThread(NULL, 0, processFileThread, filePair, 0, NULL);

        if (hThread == NULL) {

            std::cerr << "Error creating thread for file '" << inputFilename << "'." << std::endl;

            delete filePair; // Clean up memory in case of thread creation failure

        } else {

            if (!SetThreadPriority(hThread, THREAD\_PRIORITY\_ABOVE\_NORMAL)) {

                std::cerr << "Failed to set thread priority for file '" << inputFilename << "'." << std::endl;

            }

            threadHandles.push\_back(hThread); // Store thread handle

        }

    }

    // Wait for all threads to complete

    WaitForMultipleObjects(threadHandles.size(), threadHandles.data(), TRUE, INFINITE);

    // Close thread handles

    for (auto &hThread : threadHandles) {

        CloseHandle(hThread);

    }

    endTime = clock();

    writeToTextFile("file\_processing\_times\_c.txt");

    DeleteCriticalSection(&cs); // Clean up critical section

    return 0;

}

# РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММ

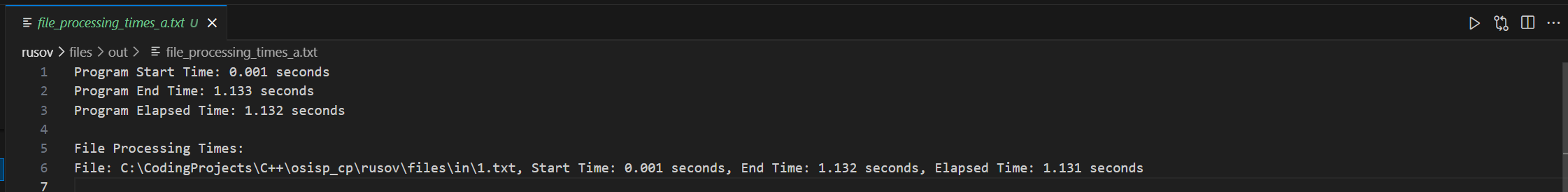


Рисунок 1 – Работа программы AH с 1 текстовым файлом

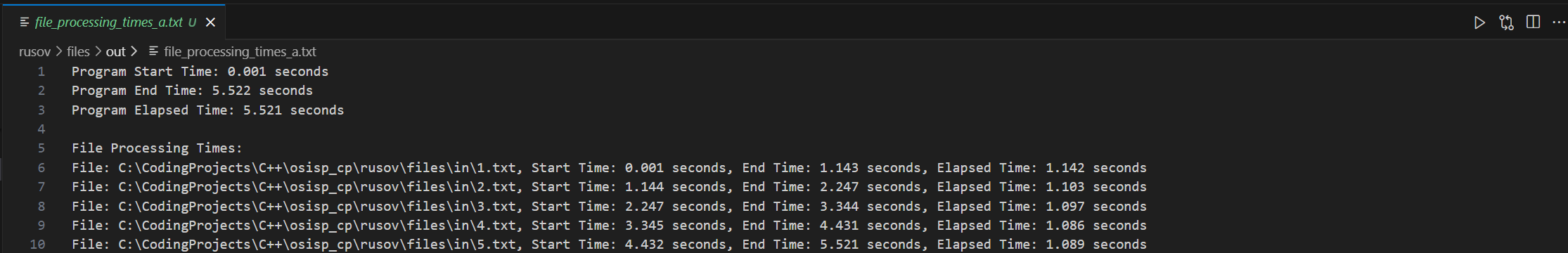


Рисунок 2 – Работа программы AH с 5 текстовыми файлами

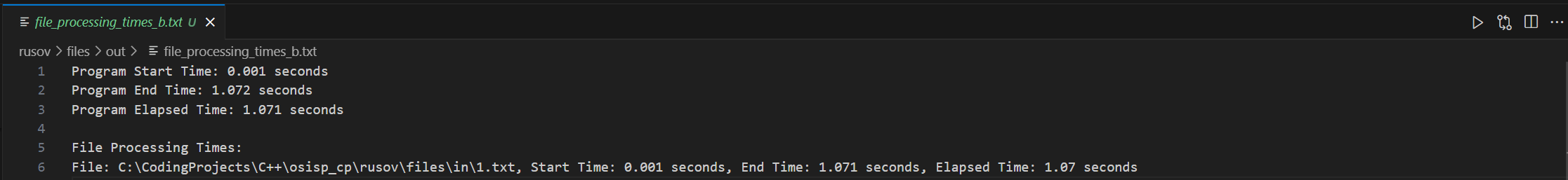


Рисунок 3 – Работа программы BH с 1 текстовым файлом

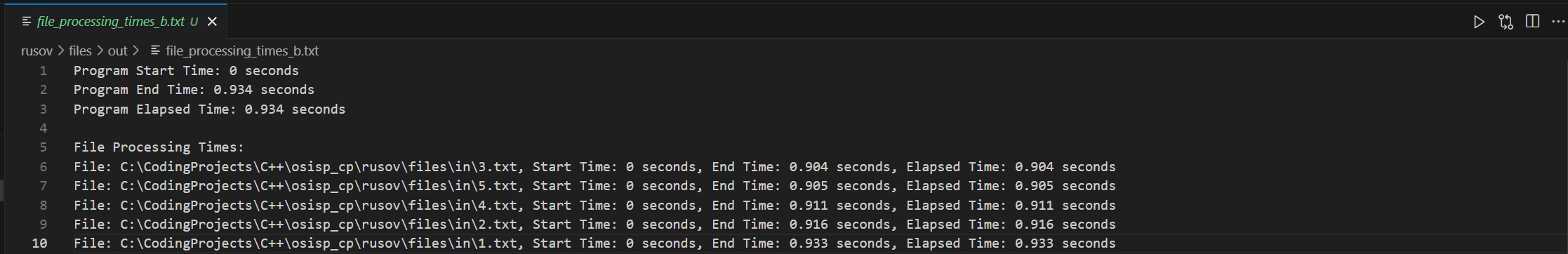


Рисунок 4 – Работа программы BH с 5 текстовыми файлами

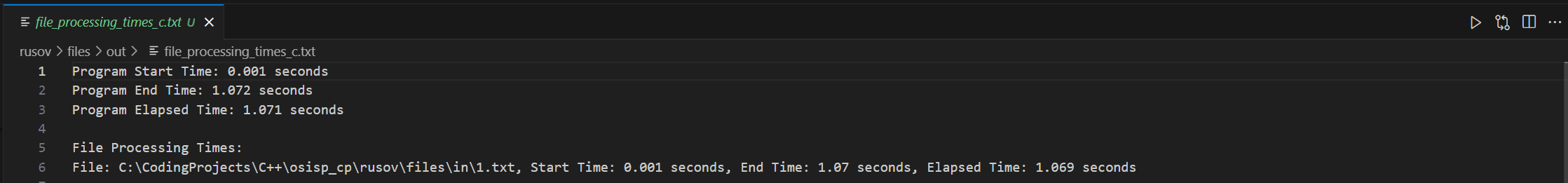


Рисунок 5 – Работа программы СH с 1 текстовым файлом

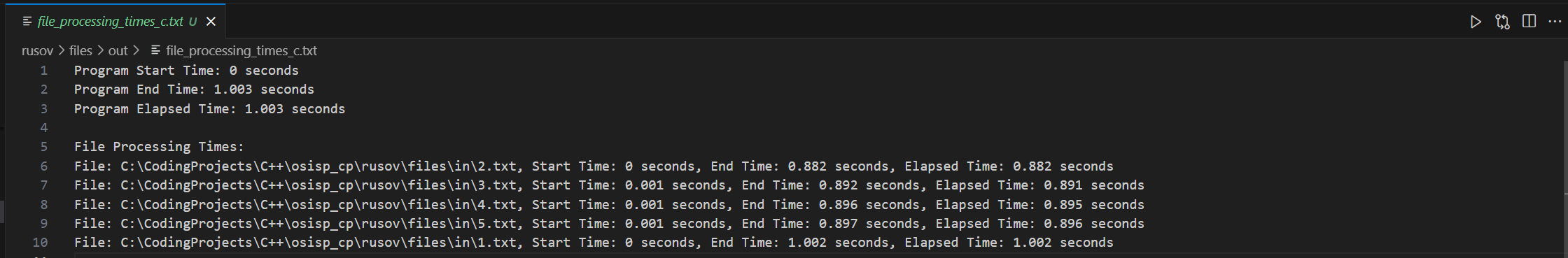


Рисунок 6– Работа программы CH с 5 текстовыми файлами

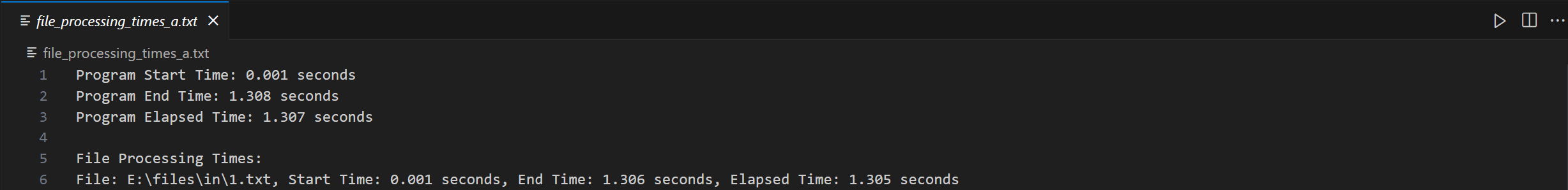


Рисунок 7 – Работа программы AF с 1 текстовым файлом

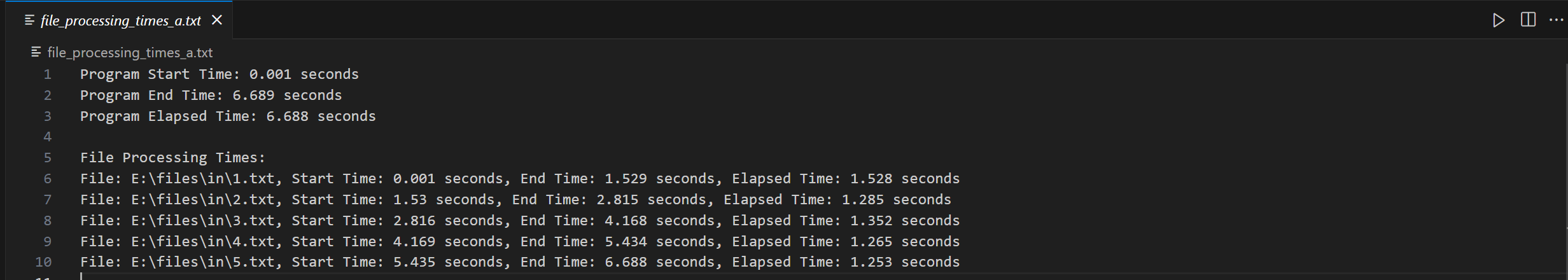


Рисунок 8 – Работа программы AF с 5 текстовыми файлами

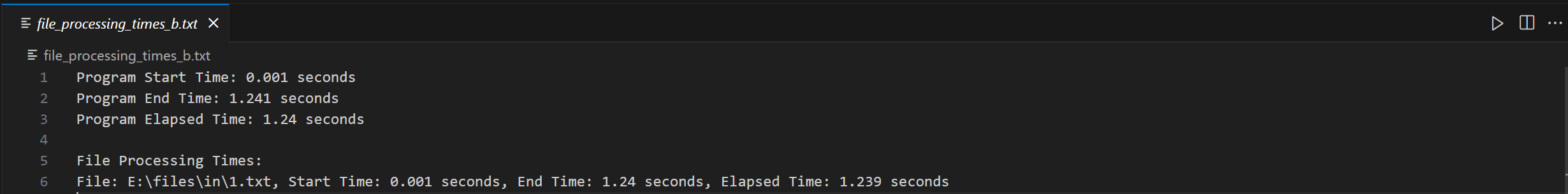


Рисунок 9 – Работа программы BF с 1 текстовым файлом

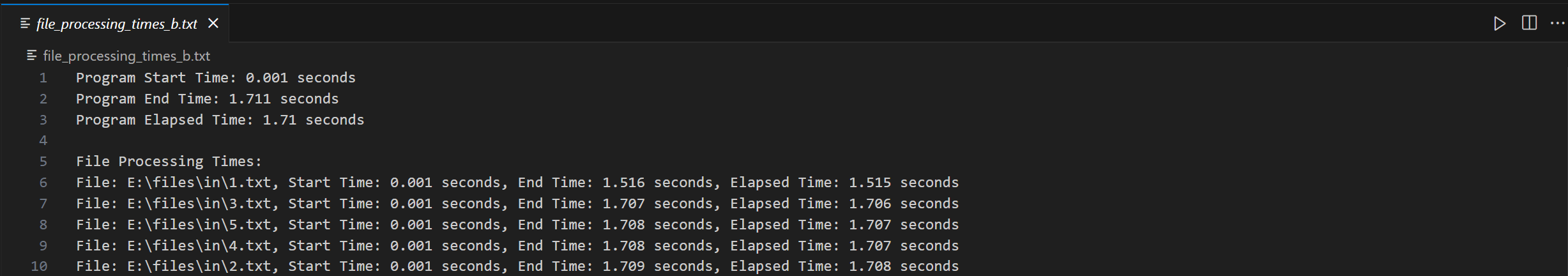


Рисунок 10 – Работа программы BF с 5 текстовыми файлами

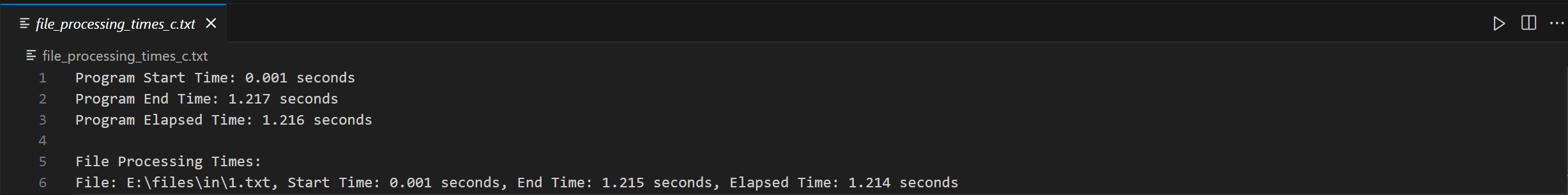


Рисунок 11 – Работа программы CF с 1 текстовым файлом

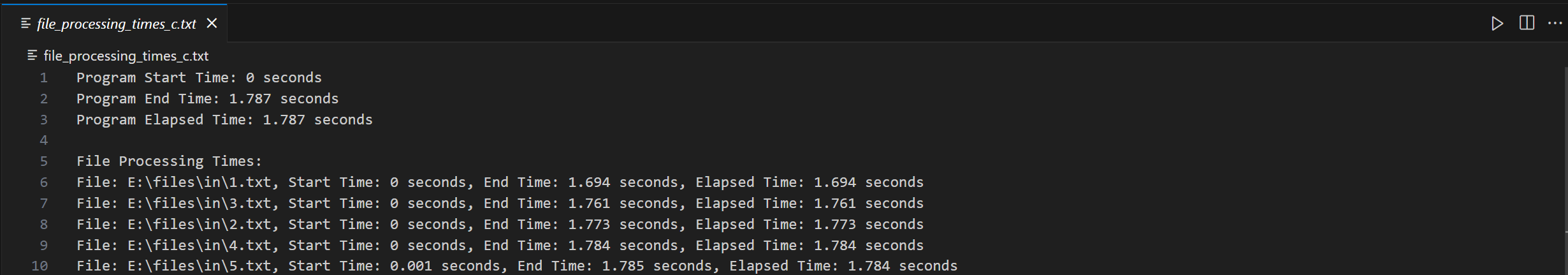


Рисунок 12 – Работа программы CF с 5 текстовыми файлами

# СКРИНШОТЫ ОКНА УТИЛИТЫ PROCESS EXPLORER

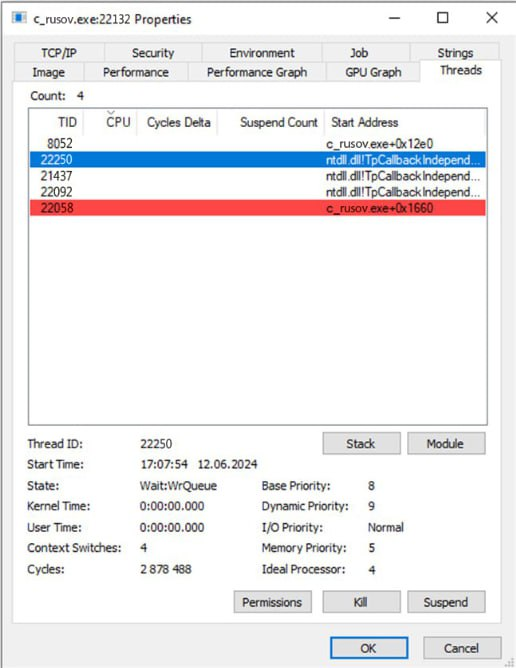


Рисунок 13 – Дочерние потоки программы С

**Base Priority (Базовый приоритет)**: Это начальный приоритет потока, установленный при его создании. Он может варьироваться от 1 (самый низкий) до 31 (самый высокий) для пользовательских процессов в Windows. Базовый приоритет потока определяется на основе приоритета процесса, к которому он принадлежит, и может быть изменен функциями управления задачами. В данном случае базовый приоритет равен 8, что соответствует уровню "Normal" (нормальный).

**Dynamic Priority (Динамический приоритет)**: Это текущий приоритет потока, который может изменяться системой в зависимости от различных факторов, таких как использование процессора, наличие ввода/вывода и другие. Система Windows может временно повышать или понижать приоритеты потоков для оптимизации работы. Например, если поток долгое время не получал процессорного времени, его приоритет может быть временно повышен. В данном случае динамический приоритет равен 9, что означает, что система временно повысила приоритет потока для улучшения производительности.

# ОПИСАНИЕ И РЕЗУЛЬТАТЫ ПРОВЕДЕННОГО ЭКСПЕРИМЕНТА

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Режим | N | t1 | t2 | t3 | t4 | t5 | tср |
| AH | 1 | 1143 | 1029 | 1198 | 1098 | 1168 | 1127,2 |
|  | 2 | 2251 | 2030 | 2361 | 2156 | 2320 | 2223,6 |
|  | 3 | 3356 | 3020 | 3470 | 3305 | 3620 | 3354,2 |
|  | 4 | 4424 | 3982 | 4547 | 4623 | 4745 | 4464,2 |
|  | 5 | 5170 | 4738 | 5146 | 4855 | 5287 | 5039,2 |
|  | 6 | 6650 | 6195 | 6510 | 6740 | 6885 | 6596 |
|  | 7 | 7737 | 6940 | 7801 | 7920 | 8060 | 7691,6 |
|  | 8 | 8659 | 7853 | 8775 | 8540 | 8791 | 8523,6 |
|  | 9 | 9725 | 8935 | 9700 | 9840 | 10100 | 9660 |
|  | 10 | 10938 | 9864 | 11153 | 10800 | 11080 | 10767 |
| BH | 1 | 1054 | 950 | 1131 | 1080 | 1010 | 1045 |
|  | 2 | 1014 | 950 | 1060 | 1020 | 1030 | 1014,8 |
|  | 3 | 1007 | 900 | 1035 | 1005 | 1100 | 1009,4 |
|  | 4 | 986 | 900 | 970 | 950 | 1000 | 961,2 |
|  | 5 | 1015 | 920 | 1080 | 1025 | 1000 | 1008 |
|  | 6 | 1002 | 900 | 1015 | 1035 | 1020 | 994,4 |
|  | 7 | 992 | 920 | 1045 | 975 | 1010 | 988,4 |
|  | 8 | 988 | 950 | 1020 | 1010 | 970 | 987,6 |
|  | 9 | 1035 | 940 | 1060 | 1010 | 1000 | 1009 |
|  | 10 | 1031 | 990 | 1080 | 1005 | 1015 | 1024,2 |
| CH | 1 | 1018 | 910 | 1100 | 1040 | 1080 | 1029,6 |
|  | 2 | 998 | 900 | 1015 | 1020 | 1000 | 986,6 |
|  | 3 | 945 | 860 | 990 | 970 | 930 | 939 |
|  | 4 | 984 | 930 | 1010 | 1020 | 970 | 982,8 |
|  | 5 | 968 | 910 | 1015 | 990 | 1040 | 984,6 |
|  | 6 | 1001 | 920 | 1015 | 1040 | 1020 | 999,2 |
|  | 7 | 1011 | 940 | 1025 | 1000 | 1060 | 1007,2 |
|  | 8 | 988 | 940 | 1020 | 1010 | 990 | 989,6 |
|  | 9 | 1007 | 930 | 1025 | 1015 | 1030 | 1001,4 |
|  | 10 | 1008 | 920 | 1020 | 1015 | 990 | 990,6 |
| AF | 1 | 1235 | 1150 | 1300 | 1200 | 1250 | 1227 |
|  | 2 | 2611 | 2400 | 2750 | 2550 | 2700 | 2602,2 |
|  | 3 | 3554 | 3300 | 3680 | 3400 | 3600 | 3506,8 |
|  | 4 | 4776 | 4500 | 4900 | 4700 | 4800 | 4735,2 |
|  | 5 | 5677 | 5400 | 5800 | 5500 | 5700 | 5615,4 |
|  | 6 | 7012 | 6700 | 7100 | 6800 | 6900 | 6902,4 |
|  | 7 | 8211 | 7800 | 8300 | 8000 | 8100 | 8082,2 |
|  | 8 | 9111 | 8700 | 9200 | 8900 | 9000 | 8982,2 |
|  | 9 | 10211 | 9500 | 10400 | 10100 | 10300 | 10102 |
|  | 10 | 11587 | 11000 | 11700 | 11400 | 11600 | 11457 |
| BF | 1 | 1416 | 1300 | 1500 | 1400 | 1450 | 1413,2 |
|  | 2 | 1603 | 1500 | 1650 | 1550 | 1580 | 1576,6 |
|  | 3 | 1505 | 1400 | 1550 | 1450 | 1600 | 1501 |
|  | 4 | 1654 | 1550 | 1700 | 1600 | 1620 | 1624,8 |
|  | 5 | 1782 | 1650 | 1820 | 1750 | 1790 | 1758,4 |
|  | 6 | 1829 | 1700 | 1850 | 1800 | 1810 | 1797,8 |
|  | 7 | 1708 | 1600 | 1750 | 1650 | 1680 | 1677,6 |
|  | 8 | 1852 | 1750 | 1900 | 1800 | 1820 | 1824,4 |
|  | 9 | 2849 | 2700 | 2900 | 2800 | 2750 | 2799,8 |
|  | 10 | 2409 | 2300 | 2500 | 2450 | 2350 | 2401,8 |
| CF | 1 | 1264 | 1150 | 1300 | 1200 | 1250 | 1232,8 |
|  | 2 | 1456 | 1350 | 1500 | 1400 | 1480 | 1437,2 |
|  | 3 | 1501 | 1400 | 1550 | 1450 | 1600 | 1500,2 |
|  | 4 | 1642 | 1550 | 1680 | 1600 | 1620 | 1618,4 |
|  | 5 | 1707 | 1600 | 1750 | 1650 | 1720 | 1685,4 |
|  | 6 | 1815 | 1700 | 1850 | 1750 | 1800 | 1783 |
|  | 7 | 2001 | 1900 | 2050 | 1950 | 1980 | 1976,2 |
|  | 8 | 1900 | 1800 | 1950 | 1850 | 1920 | 1884 |
|  | 9 | 2767 | 2600 | 2800 | 2700 | 2650 | 2703,4 |
|  | 10 | 2398 | 2300 | 2450 | 2350 | 2400 | 2379,6 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Режим | Файлы на жестком диске (H) | | | | | | | | | |
| N=1 | N=2 | N=3 | N=4 | N=5 | N=6 | N=7 | N=8 | N=9 | N=10 |
| A | 1127,2 | 2223,6 | 3354,2 | 4464,2 | 5039,2 | 6596 | 7691,6 | 8523,6 | 9660 | 10767 |
| B | 1045 | 1014,8 | 1009,4 | 961,2 | 1008 | 994,4 | 988,4 | 987,6 | 1009 | 1024,2 |
| C | 1029,6 | 986,6 | 939 | 982,8 | 984,6 | 999,2 | 1007,2 | 989,6 | 1001,4 | 990,6 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Файлы на flash-накопителе (F) | | | | | | | | | |
| N=1 | N=2 | N=3 | N=4 | N=5 | N=6 | N=7 | N=8 | N=9 | N=10 |
| 1227 | 2602,2 | 3506,8 | 4735,2 | 5615,4 | 6902,4 | 8082,2 | 8982,2 | 10211 | 11457 |
| 1413,2 | 1576,6 | 1501 | 1624,8 | 1758,4 | 1797,8 | 1677,6 | 1824,4 | 2799,8 | 2401,8 |
| 1232,8 | 1437,2 | 1500,2 | 1618,4 | 1685,4 | 1783 | 1976,2 | 1884 | 2703,4 | 2379,6 |

# ДИАГРАММЫ ТРАСС ПОТОКОВ

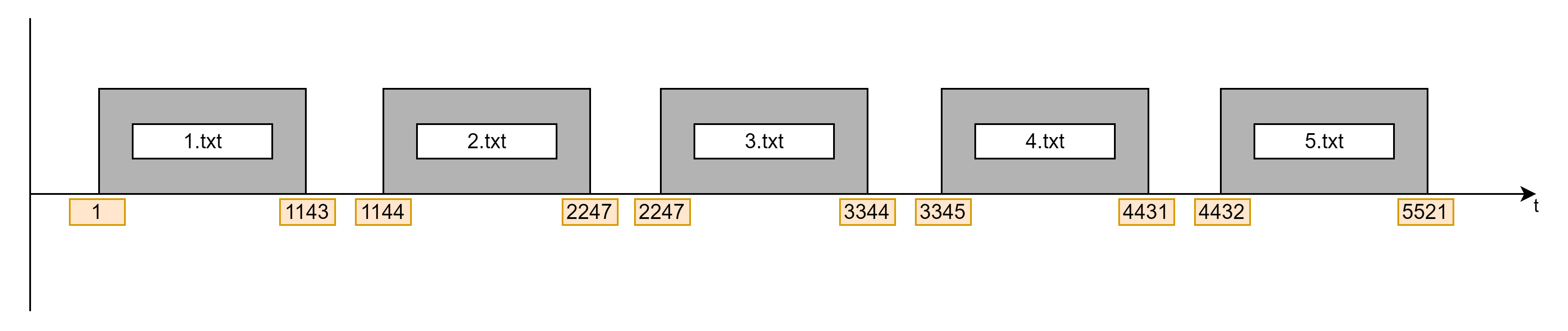


Рисунок 14 – Диаграмма трасс потоков программы AH с 5 текстовыми файлами

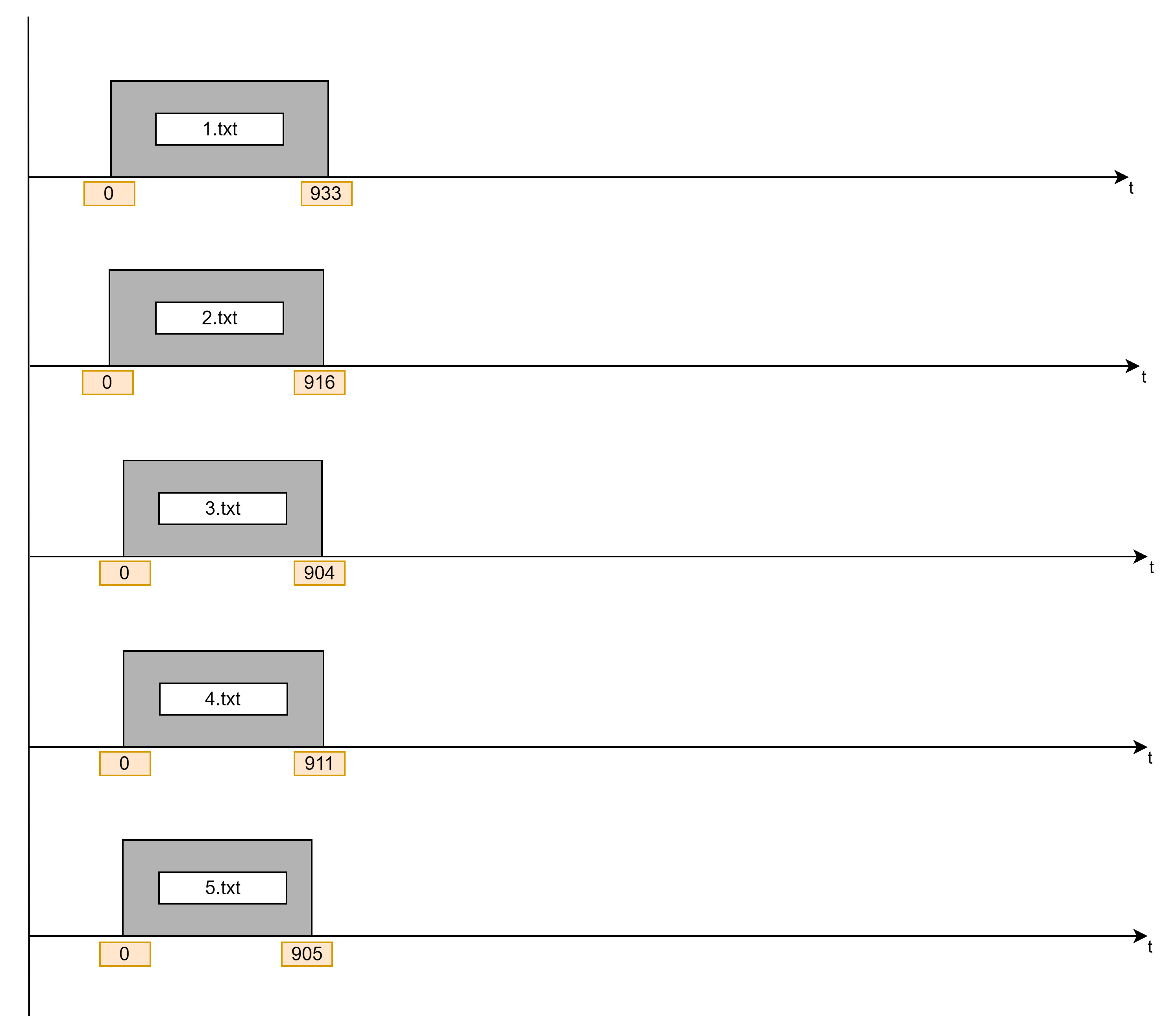


Рисунок 15 – Диаграмма трасс потоков программы BH с 5 текстовыми файлами

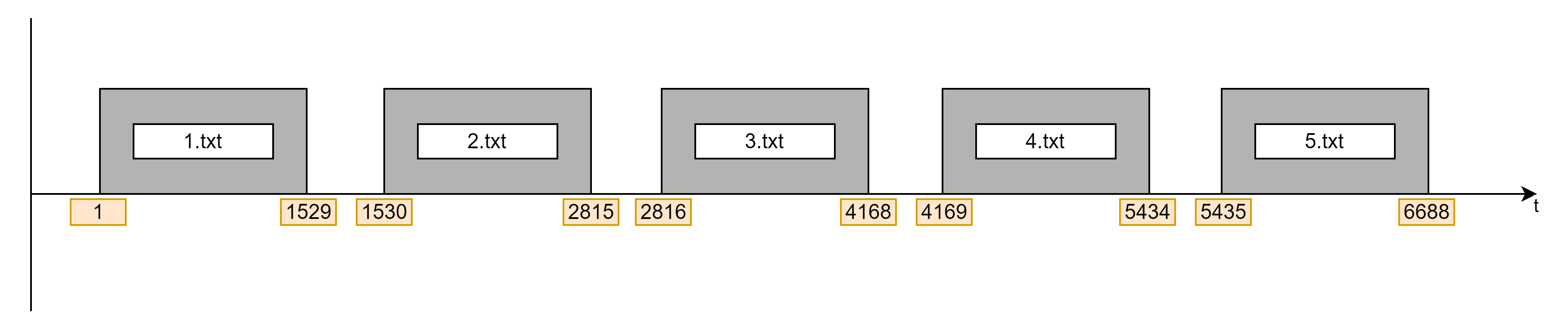


Рисунок 16 – Диаграмма трасс потоков программы AF с 5 текстовыми файлами

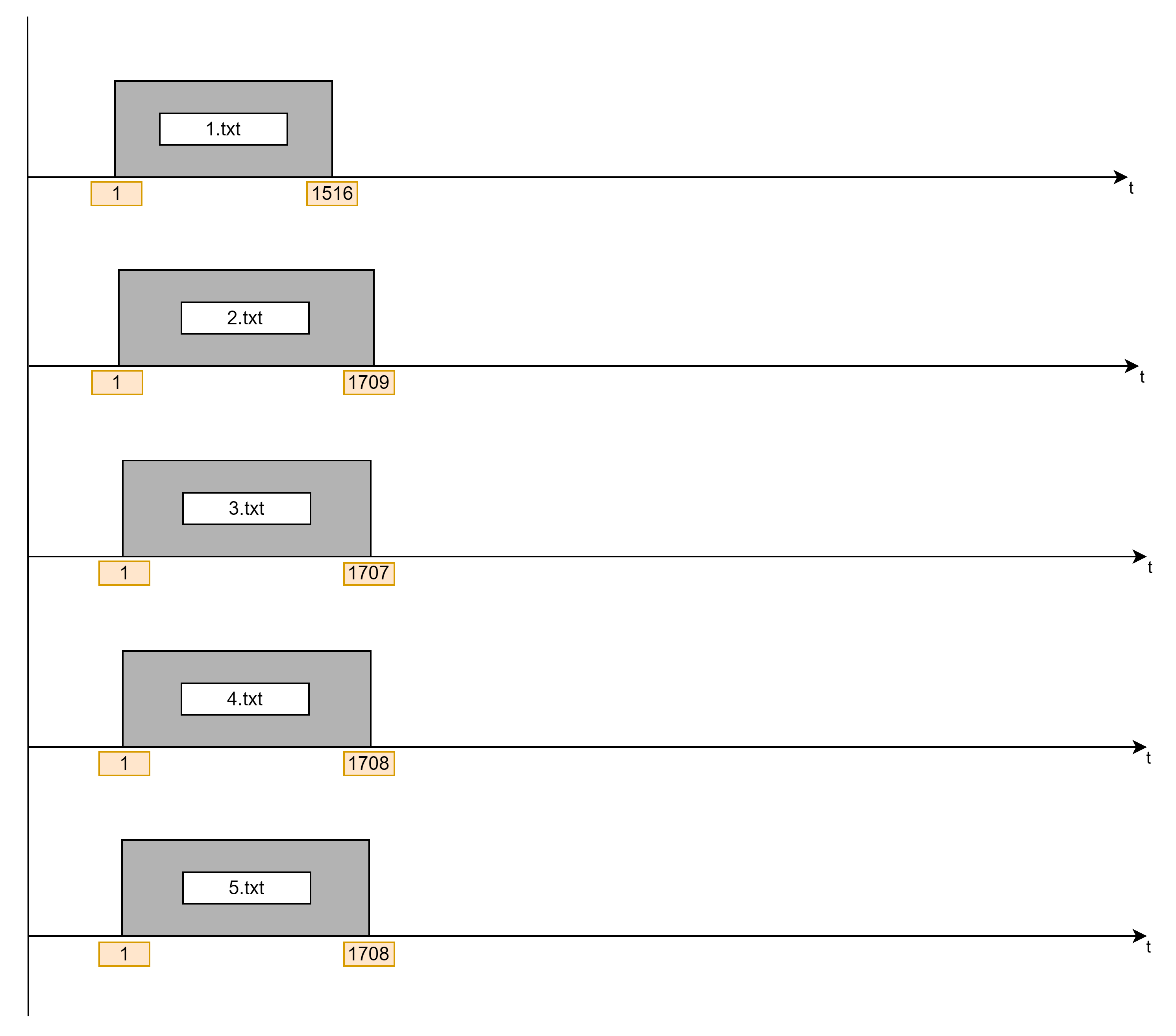


Рисунок 17 – Диаграмма трасс потоков программы BF с 5 текстовыми файлами

# ВЫВОДЫ ПО РЕЗУЛЬТАТАМ ПРОВЕДЕННГО ЭКСПЕРИМЕНТА

**Сравнение многопоточного режима (B) с однопоточным режимом (A) при размещении файлов на жестком диске (H)**:

Многопоточная обработка файлов (режим B) на жестком диске показала значительно более высокую производительность по сравнению с однопоточной (режим A). Среднее время обработки файлов в многопоточном режиме было намного ниже на каждом этапе, особенно при увеличении количества файлов. Это объясняется тем, что многопоточность позволяет параллельно обрабатывать файлы, эффективно используя ресурсы процессора и дисковой системы.

**Сравнение среднего времени обработки в многопоточном режиме (B) по сравнению с режимом (A) при использовании съемного flash-накопителя (F)**:

При использовании съемного flash-накопителя многопоточный режим также показал лучшие результаты по сравнению с однопоточным. Однако разница в производительности была менее значительной по сравнению с жестким диском, что можно объяснить более медленной скоростью чтения/записи на flash-накопителе. **Однако при увеличении количества файлов в обработке время выполнения в многопоточном режиме на flash-накопителе начинает расти быстрее, чем на жестком диске (H), из-за ограниченной пропускной способности и задержек при доступе к данным на flash-накопителе.**

**Объяснение с помощью диаграмм трасс потоков, каким образом многопоточность повлияла на общее время обработки файлов:**

Диаграммы трасс потоков показывают, что в многопоточном режиме обработка нескольких файлов происходит одновременно, что сокращает общее время выполнения программы. В однопоточном режиме файлы обрабатываются последовательно, что увеличивает время обработки при большом объеме данных.

**Повышение приоритета потоков и его влияние на производительность обработки**:

В эксперименте с повышением приоритета потоков (режим C) было замечено небольшое улучшение производительности. Потоки с повышенным приоритетом получали больше процессорного времени, что приводило к уменьшению времени обработки некоторых файлов, особенно при высокой загруженности системы.

**Влияние количества ядер процессора на результаты**:

Многопоточный режим (B и C) особенно эффективно использует многопроцессорные системы. Чем больше ядер у процессора, тем лучше распределяются задачи между потоками, что сокращает общее время обработки файлов. Однопоточный режим не способен эффективно использовать все ядра процессора, что замедляет выполнение программы.

**Необходимость использования средств синхронизации потоков**:

В многопоточном режиме синхронизация потоков была важна для предотвращения конфликтов при доступе к общим ресурсам (например, файлам). Использование критических секций или других механизмов синхронизации гарантировало корректную работу программы без ошибок, связанных с параллельной обработкой.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Таненбаум Э., Бос Х. Современные операционные системы. - 4-е изд. - СПб.: Питер, 2015. - 1120 с.
2. Варфоломеев В.А. Организация многопоточных приложений в ОС Windows. Учебно-методическое пособие. — М.: МИИТ, 2024.— 24 с.
3. Соларес С. Windows System Programming. - 4-е изд. - Addison-Wesley, 2007. - 976 с.
4. Рихтер Д. Программирование под Windows. - М.: Диалектика, 2011. - 1360 с.
5. Гук М. Параллельное программирование для многопроцессорных систем. - М.: ДМК Пресс, 2014. - 400 с.
6. Microsoft. Multithreading and Concurrency: Windows Developer Documentation [Электронный ресурс] // Microsoft Docs. - 2023. URL: <https://docs.microsoft.com/en-us/windows/win32/procthread/multithreading-and-concurrency> (дата обращения: 12.06.2024).
7. Вейс М. Параллельное программирование с использованием C++. - М.: Вильямс, 2016. - 768 с.