Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет

информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра Информатики

Дисциплина «Операционные среды и системное программирование»

**ОТЧЕТ**

к лабораторной работе №1

на тему:

**«ЦЕЛОЧИСЛЕННЫЕ АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ. ОБРАБОТКА МАССИВОВ ЧИСЛОВЫХ ДАННЫХ»**

БГУИР 6-05-0612-02 73

|  |
| --- |
| Выполнила студентка группы 353504  ЛЕБЕДЕВА Милана Валерьевна |
|  |
| (дата, подпись студента) |
| Проверил ассистент каф. Информатики  ГРИЦЕНКО Никита Юрьевич |
|  |
| (дата, подпись преподавателя) |

Минск 2025

# 1 Цель работы

Возобновление, закрепление и развитие навыков программирования приложений Windows. Концепции вычислительных процессов, потоков, нитей; реализация в Windows. Основные этапы жизненного цикла процессов (потоков) и элементарное управление ими: порождение, завершение, получение и изменение состояния. Типичное (простое) использование многозадачности и многопоточности.

# 2 задание

**Задание 1. Вариант 7.** Чтение файла несколькими потоками, «сборка» результата. Количество потоков, файл для чтения – выбор пользователем. Количество потоков – аналогично предыдущему. Оценка времени, зависимость от начальных параметров.

# 3 выполнение работы

Для реализации задания был использован язык программирования C++. Было разработано многопоточное консольное приложение для анализа текстового файла. Приложение считывает указанный пользовательский файл, разделяет содержимое файла на строки и распределяет их между заданным количеством потоков для обработки. Каждый поток подсчитывает количество строк в своей части данных, количество символов и частоту встречаемости каждого слова. Работа потоков синхронизируется с помощью технологии мьютекс из стандартной библиотеки «mutex». Ниже представлен листинг кода программы.

#include <iostream>#include <fstream>#include <vector>

#include <string>#include <windows.h>#include <algorithm>#include <map>

#include <mutex>#include <sstream>#include <cctype>#include <chrono>

using namespace std;

//мьютекс для синхронизации вывода

mutex output\_mutex;

enum CountMode {

VISIBLE\_CHARS,

ALL\_CHARS

};

struct ThreadData {

vector<string> lines;

int thread\_id = 0;

int line\_count = 0;

int char\_count = 0;

map<string, int> word\_count;

CountMode count\_mode = VISIBLE\_CHARS;

bool is\_last\_global\_line = false;

bool file\_ends\_with\_newline = false;

};

vector<string> splitIntoWords(const string& line) {

vector<string> words;

stringstream ss(line);

string word;

while (ss >> word) {

string cleaned\_word;

for (char c : word) {

if (isalnum(static\_cast<unsigned char>(c))) {

cleaned\_word += tolower(static\_cast<unsigned char>(c));

}

}

if (!cleaned\_word.empty()) {

words.push\_back(cleaned\_word);

}

}

return words;

}

DWORD WINAPI ProcessLines(LPVOID lpParam) {

ThreadData\* data = (ThreadData\*)lpParam;

data->line\_count = 0;

data->char\_count = 0;

for (size\_t idx = 0; idx < data->lines.size(); ++idx) {

const string& line = data->lines[idx];

data->line\_count++;

if (data->count\_mode == ALL\_CHARS) {

data->char\_count += static\_cast<int>(line.length());

bool is\_last\_line\_of\_thread = (idx == data->lines.size() - 1);

if (!(data->is\_last\_global\_line && is\_last\_line\_of\_thread && !data->file\_ends\_with\_newline)) {

data->char\_count++;

}

}

else {

data->char\_count += static\_cast<int>(line.length());

}

// Разделяем строку на слова

vector<string> words = splitIntoWords(line);

for (const string& word : words) {

data->word\_count[word]++;

}

}

{

lock\_guard<mutex> lock(output\_mutex);

cout << "Поток " << data->thread\_id

<< " обработал " << data->lines.size() << " строк" << endl;

}

return 0;

}

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

setlocale(LC\_ALL, "RUSSIAN");

cout << "=== Многопоточный анализатор файлов ===" << endl;

if (argc < 3 || argc > 4) {

cout << "Использование: myfiletool.exe <filename> <thread\_count> [all|visible]" << endl;

return 1;

}

string filename = argv[1];

int thread\_count = atoi(argv[2]);

CountMode count\_mode = VISIBLE\_CHARS;

if (argc == 4) {

string mode = argv[3];

if (mode == "all") {

count\_mode = ALL\_CHARS;

cout << "Режим: все символы (с учётом служебных)" << endl;

}

else if (mode == "visible") {

count\_mode = VISIBLE\_CHARS;

cout << "Режим: только видимые символы" << endl;

}

else {

cout << "Ошибка: неизвестный режим \"" << mode << "\". Используйте all или visible." << endl;

return 1;

}

}

else {

cout << "Режим: только видимые символы (по умолчанию)" << endl;

}

if (thread\_count <= 0) {

cout << "Ошибка: количество потоков должно быть положительным числом." << endl;

return 1;

}

ifstream file(filename);

if (!file.is\_open()) {

cout << "Ошибка: не могу открыть файл " << filename << endl;

return 1;

}

vector<string> all\_lines;

string line;

while (getline(file, line)) {

all\_lines.push\_back(line);

}

file.close();

int total\_lines = static\_cast<int>(all\_lines.size());

cout << "Прочитано строк: " << total\_lines << endl;

cout << "Количество потоков: " << thread\_count << endl;

if (total\_lines == 0) {

cout << "Файл пуст." << endl;

return 0;

}

bool file\_ends\_with\_newline = false;

{

ifstream check(filename, ios::binary);

if (check.is\_open()) {

check.seekg(0, ios::end); //p->end

if (check.tellg() > 0) {//p: length

check.seekg(-1, ios::end);

char last\_char;

check.get(last\_char);

file\_ends\_with\_newline = (last\_char == '\n');

}

check.close();

}

}

if (thread\_count > total\_lines) {

thread\_count = total\_lines;

cout << "Внимание: потоков больше, чем строк. Использую " << thread\_count << " поток(а)." << endl;

}

vector<ThreadData> thread\_data(thread\_count); //data of threads

vector<HANDLE> threads(thread\_count); //descriptors of threads

int lines\_per\_thread = total\_lines / thread\_count;

int remainder = total\_lines % thread\_count;

int line\_index = 0;

auto start\_time = chrono::high\_resolution\_clock::now();

for (int i = 0; i < thread\_count; i++) {

int lines\_for\_this\_thread = lines\_per\_thread + (i < remainder ? 1 : 0);

for (int j = 0; j < lines\_for\_this\_thread; j++) {

thread\_data[i].lines.push\_back(all\_lines[line\_index++]);

}

thread\_data[i].thread\_id = i + 1;

thread\_data[i].count\_mode = count\_mode;

if (i == thread\_count - 1) {

thread\_data[i].is\_last\_global\_line = true;

}

thread\_data[i].file\_ends\_with\_newline = file\_ends\_with\_newline;

threads[i] = CreateThread(NULL, 0, ProcessLines, &thread\_data[i], 0, NULL);

if (threads[i] == NULL) {

cout << "Ошибка: не могу создать поток " << i + 1 << endl;

return 1;

}

{

lock\_guard<mutex> lock(output\_mutex);

cout << "Создан поток " << (i + 1)

<< " для обработки " << lines\_for\_this\_thread << " строк" << endl;

}

}

WaitForMultipleObjects(thread\_count, threads.data(), TRUE, INFINITE);

int computed\_lines = 0;

int computed\_chars = 0;

map<string, int> global\_word\_count;

for (int i = 0; i < thread\_count; i++) {

computed\_lines += thread\_data[i].line\_count;

computed\_chars += thread\_data[i].char\_count;

for (const auto& pair : thread\_data[i].word\_count) {

global\_word\_count[pair.first] += pair.second;

}

CloseHandle(threads[i]);

}

auto end\_time = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::milliseconds>(end\_time - start\_time);

cout << "\n=== РЕЗУЛЬТАТЫ АНАЛИЗА ===" << endl;

cout << "Общее количество строк: " << computed\_lines << endl;

cout << "Общее количество символов: " << computed\_chars << endl;

ifstream file\_check(filename, ios::binary | ios::ate);

if (file\_check.is\_open()) {

long long file\_size = file\_check.tellg();

file\_check.close();

cout << "Размер файла (байт): " << file\_size << endl;

}

cout << "Время выполнения: " << duration.count() << " мс" << endl;

if (!global\_word\_count.empty()) {

auto most\_common = max\_element(

global\_word\_count.begin(),

global\_word\_count.end(),

[](const pair<string, int>& a, const pair<string, int>& b) {

return a.second < b.second;

}

);

cout << "Самое частое слово: '" << most\_common->first

<< "' (" << most\_common->second << " раз)" << endl;

cout << "\nТоп-10 слов:" << endl;

vector<pair<string, int>> sorted\_words(global\_word\_count.begin(), global\_word\_count.end());

sort(sorted\_words.begin(), sorted\_words.end(),

[](const auto& a, const auto& b) { return b.second < a.second; });

int top\_n = min(10, static\_cast<int>(sorted\_words.size()));

for (int i = 0; i < top\_n; i++) {

cout << i + 1 << ") '" << sorted\_words[i].first

<< "' : " << sorted\_words[i].second << endl;

}

}

cout << "Анализ завершен" << endl;

return 0;

}

Ключевыми элементами кода является структура для передачи данных в каждый поток, глобальный мьютекс для синхронизации. Функция потока ProccesLines() является точкой входа для создаваемых потоков и принимает один параметр: указатель на структуру. Функция splitIntoWords() разбивает строку на слова, очищает их от ненужных символов и приводит к нижнему регистру.

Создание потоков реализовано в главной функции с помощью функции CreateThread() API Windows. Главный поток собирает данные из всех структур, суммируя счетчики и объединяя карты слов.

# 4 тестирование

Тест 1. Базовый функционал программы – создание, выполнение и завершение потоков.

Команда запуска: LR1.exe test1.txt 2 visible

Результаты первого теста представлены на рисунке 1.

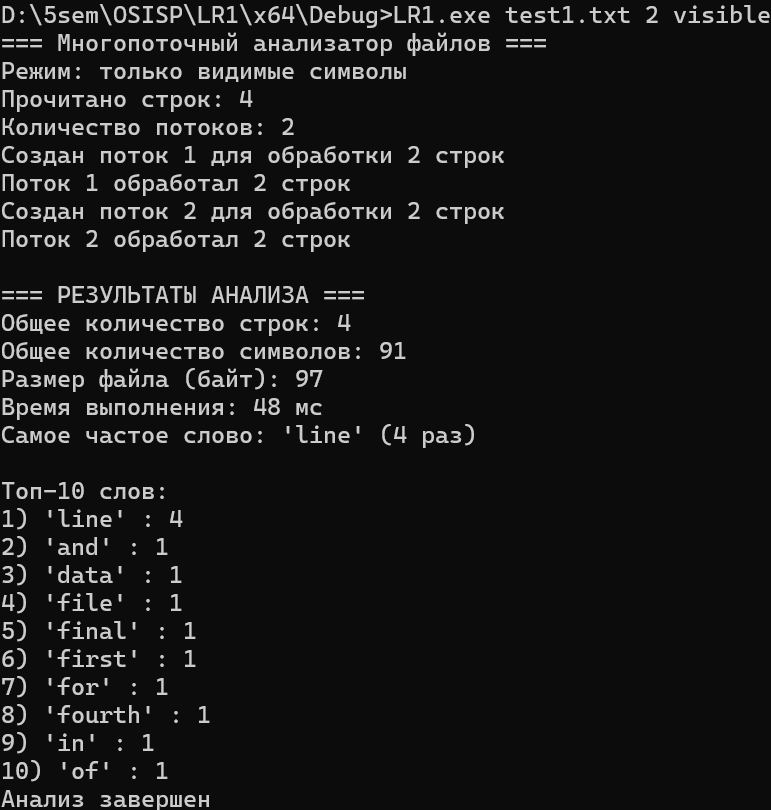


Рисунок 1 – Результаты первого теста

Результаты показали, что программа корректно создала 2 потока. Работа была распределена поровну, что соответствует алгоритму планирования. Каждый поток независимо обработал свой набор данных. Данные были обработаны полностью и правильно.

Тест 2. Краевой случай – ситуация, когда потоков больше, чем строк.

Команда запуска: LR1.exe test2.txt 5 visible

Результаты второго теста представлены на рисунке 2.

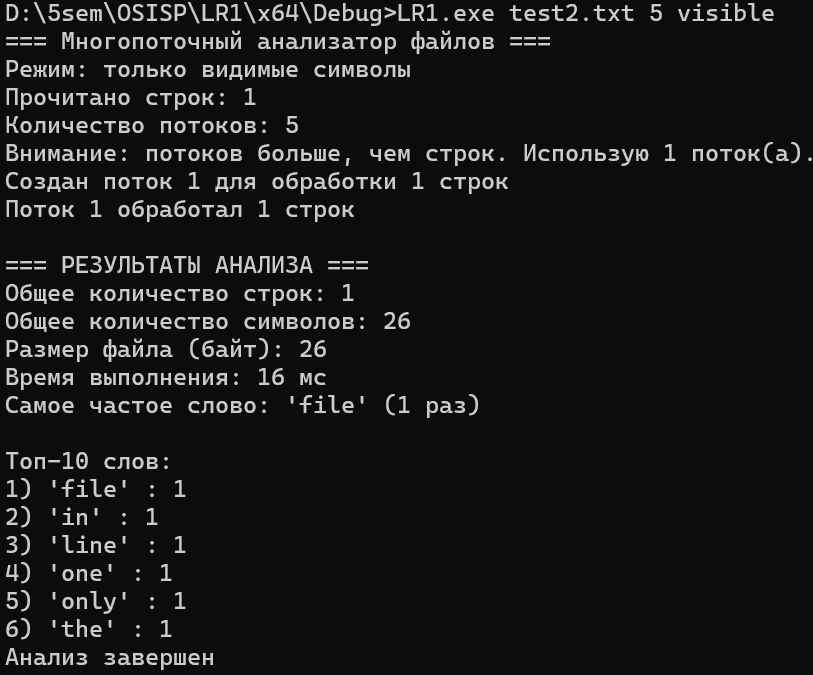


Рисунок 2 – Результаты второго теста

Результаты показали, что программа корректно определила, что запрошено больше потоков, чем строк в файле. Алгоритм автоматически скорректировал количество потоков до необходимого минимума, что предотвращает создание избыточных потоков, которые не получили бы данных для обработки. Был создан и успешно выполнен только один поток, что подтверждается соответствующими сообщениями.

Тест 3. Демонстрация синхронизации

Команда запуска: LR1.exe test3.txt 3 visible (с закомментированным мьютексом).

Результаты третьего теста – некорректного вывода представлены на рисунке 3.

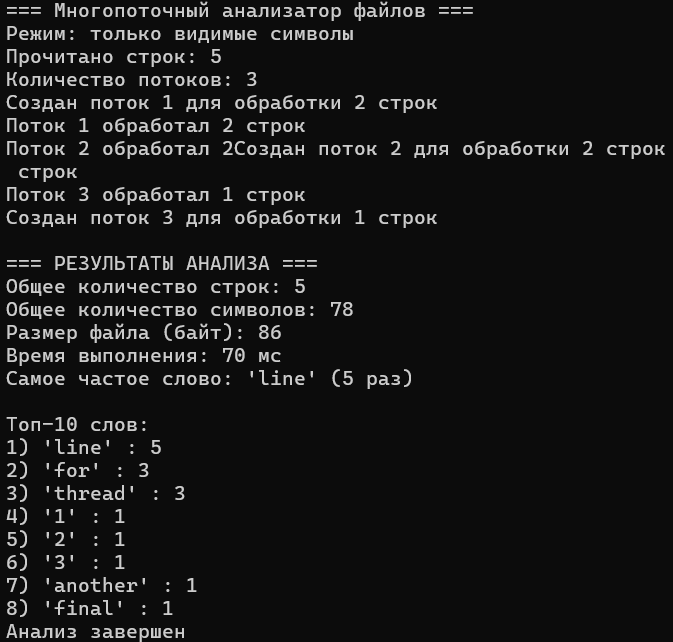


Рисунок 3 – Результаты третьего теста

Вывод программы представляет собой перемешанные символов и сообщений от разных потоков. Сообщение от потока 1 было прервано сообщением от потока 2, которое, в свою очередь, было прервано сообщением от потока 3. Этот тест наглядно демонстрирует, почему механизмы синхронизации, такие как мьютекс, являются неотъемлемой частью многопоточного программирования. Без них невозможно гарантировать целостность и порядок операций с общими ресурсами. После возвращения мьютекса вывод снова станет упорядоченным, что подтвердит правильность работы механизма синхронизации.

Тест 4. Оценка производительности.

Для проверки используется большой текстовый файл около 10 МБ.

Команда запуска: LR1.exe test4.txt 1/2/4/8 visible

Результаты четвертого теста с 1 потоком представлены на рисунке 4.

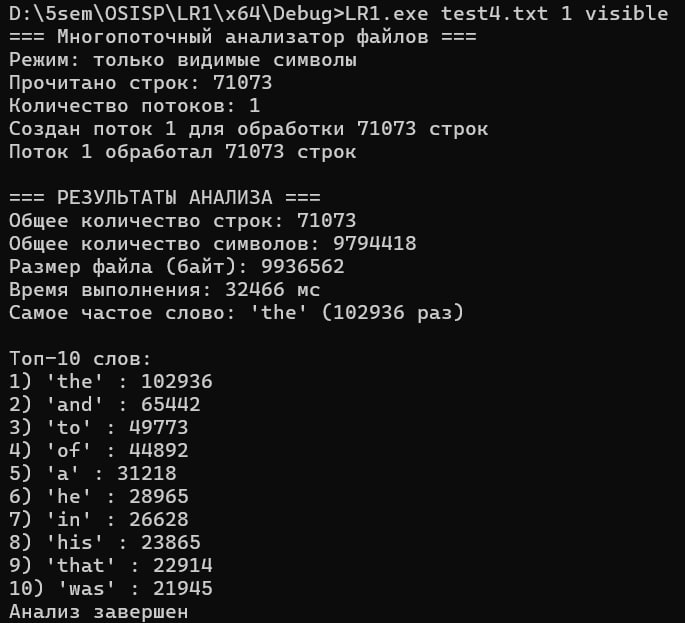


Рисунок 4 – Результаты четвертого теста с 1 потоком

Вся нагрузка легла на одно ядро, производительности которого не хватает для быстрой обработки всего файла. Это базовая, наихудшая точка отсчета.

Результаты четвертого теста с 2 потоками представлены на рисунке 5.

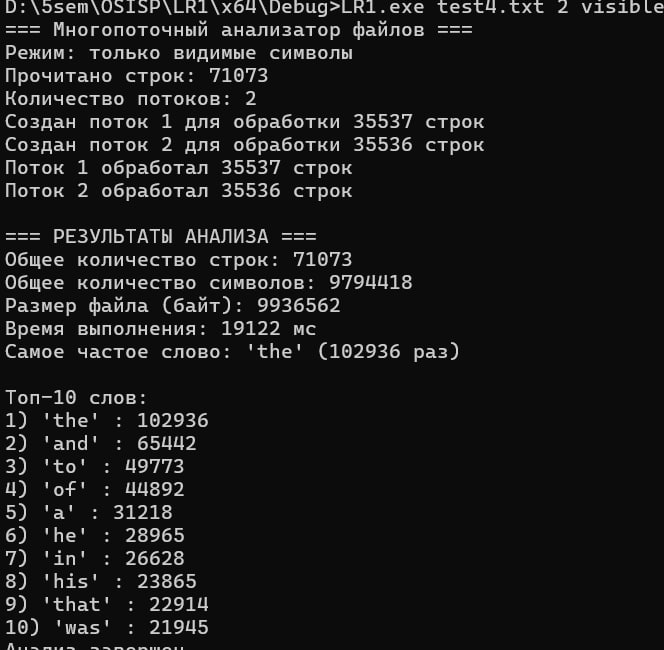


Рисунок 5 – Результаты четвертого теста с 2 потоками

В результате получается ускорение приблизительно в 1.7 раз по сравнению с одним потоком.

Результаты четвертого теста с 4 потоками представлены на рисунке 6.

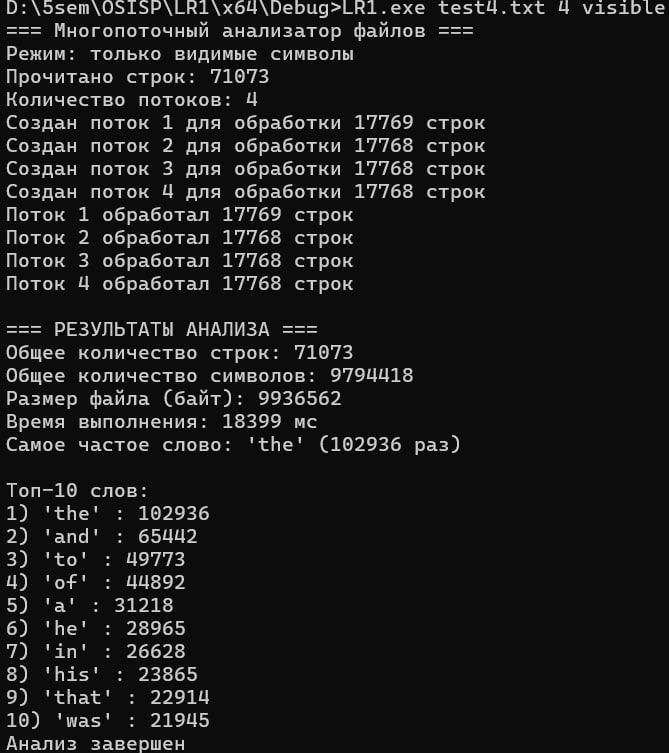


Рисунок 6 – Результаты четвертого теста с 4 потоками

В результате получается ускорение приблизительно в 1.76 раз.

Результаты четвертого теста с 8 потоками представлены на рисунке 7.

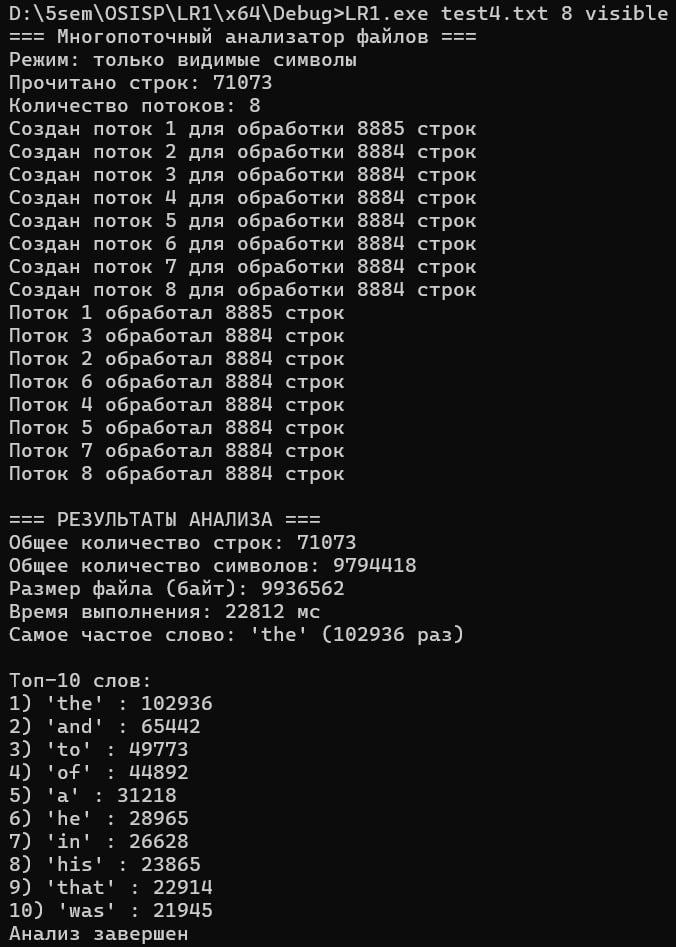


Рисунок 7 – Результаты четвертого теста с 8 потоками

В результате получается замедление относительно четырех потоков.

Было установлено, что для процессора Intel Core i5-12450H оптимальным для данной задачи является использование 4 потоков. Это объясняется тем, что данное количество позволяет планировщику операционной системы разместить каждый поток на отдельном высокопроизводительном P-ядре, обеспечивая максимальную скорость вычислений без конкуренции за ресурсы.

В случае с 8 потоками, 4 потока попадают на быстрые P-ядра. Но так как P-ядер всего 4, а потоков уже 8, на каждом P-ядре вынуждены работать по 2 потока одновременно из-за Hyper-Threading. Это создает небольшие накладные расходы из-за конкуренции за ресурсы ядра.

# Вывод

В ходе выполнения работы были освоены ключевые функции WinAPI для работы с потоками. Были освоены основные этапы жизненного цикла потока. Было также продемонстрировано типичное использование многопоточности и использован механизм мьютексов для синхронизации доступа к общему ресурсу.

Проведен анализ влияния многопоточности на производительность. Было выявлено, что производительность растет лишь до определенного предела (оптимального количества потоков), а затем может снижаться за счет накладных расходов на переключение контекста и особенностей аппаратной архитектуры (гибридных ядер современного процессора). Это наглядно иллюстрирует как преимущества, так и сложности многопоточного программирования.