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

ОТЧЁТ

к лабораторной работе №2

на тему

**РАБОТА С ФАЙЛАМИ**

Студент: Ющук И.А.

Преподаватель: Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc177507035)

[2 Описание функций программы 4](#_Toc177507036)

[2.1 Асинхронное чтение 4](#_Toc177507037)

[2.2 Обработка прочитанных данных 4](#_Toc177507038)

[2.3 Запись данных в файл 5](#_Toc177507039)

[Заключение 6](#_Toc177507040)

[Список использованных источников 7](#_Toc177507041)

[Приложение А (обязательное) Исходный код программы 8](#_Toc177507042)

1 ФОРМУЛИРОВКА ЗАДАЧИ

Реализация обработки содержимого файла данных, используя асинхронный ввод-вывод (чтение/запись очередных порций данных параллельно с выполнением обработки данных в памяти). Варьирование количества параллельно инициированных операций ввода вывода, влияние их на общею производительность. Поиск «узкого места» в программной реализации. Оценка эффективности (производительности) по сравнению с традиционным подходом (чтение → обработка → выгрузка). Опционально – то же по сравнению с многопоточной реализацией. Возможные варианты обработки: сортировка, статистическая обработка, криптография и т.д. Объем данных следует выбирать достаточно большим, чтобы длительность обработки была заметна и хорошо поддавалась измерению.

2 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Согласно формулировке задачи, были спроектированы следующие функции программы:

– функция асинхронного чтения данных из файла;

– функция обработка прочитанных данных;

– функция асинхронного ввода обработанных данных в файл.

## **2.1 Асинхронное чтение**

При запуске программы открывается заданный текстовый файл и из него несколько потоков асинхронно считывают данные. [1,2] (рисунок 2.1).

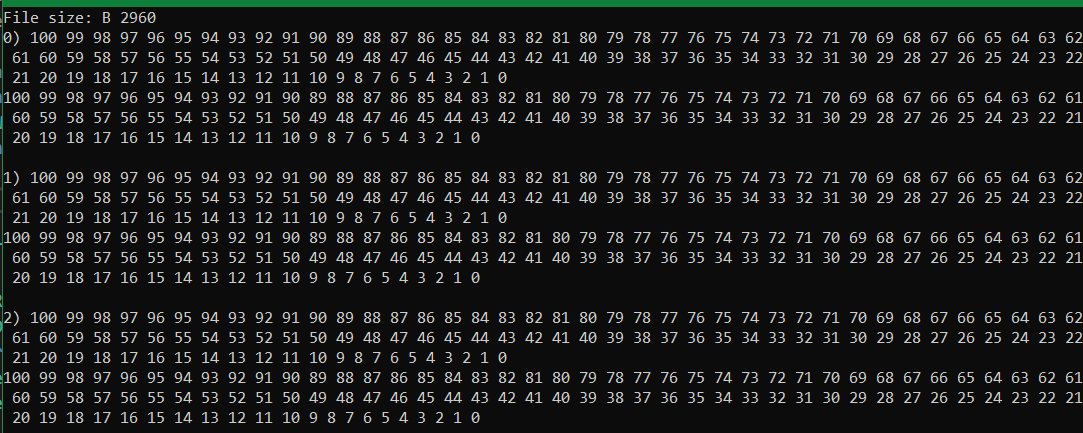
****

Рисунок 2.1 – Результат асинхронного чтения файла

## **2.2** **Обработка прочитанных данных**

После чтения файла буфер каждого потока проходит обработку. Сначала буфер разделяется на отдельные строки, затем каждая строка превращается в целочисленный вектор. После вектор сортируется с помощью пузырьковой сортировки. В конце обработки целочисленные векторы превращаются обратно в указатель и сохраняется в буфер. (рисунок 2.2).

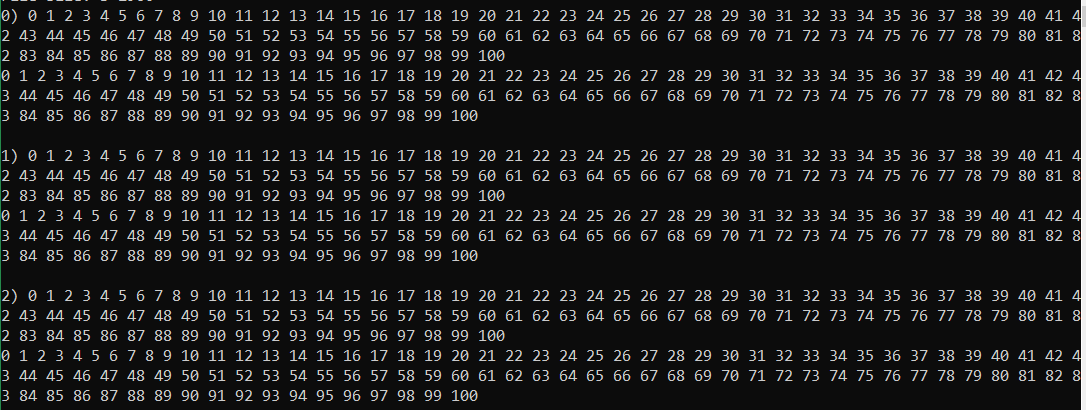


Рисунок 2.2 – Результат обработки и сортировки данных

## **2.3 Запись данных в файл**

После обработки данные асинхронно записываются в заданный файл. [3] В заранее заданный файл записывается результат. Каждый поток из своего же буфера записывает данные с тем же смещением что и считывал. [4] Таким образом общая структура файла сохранится (рисунок 2.3).

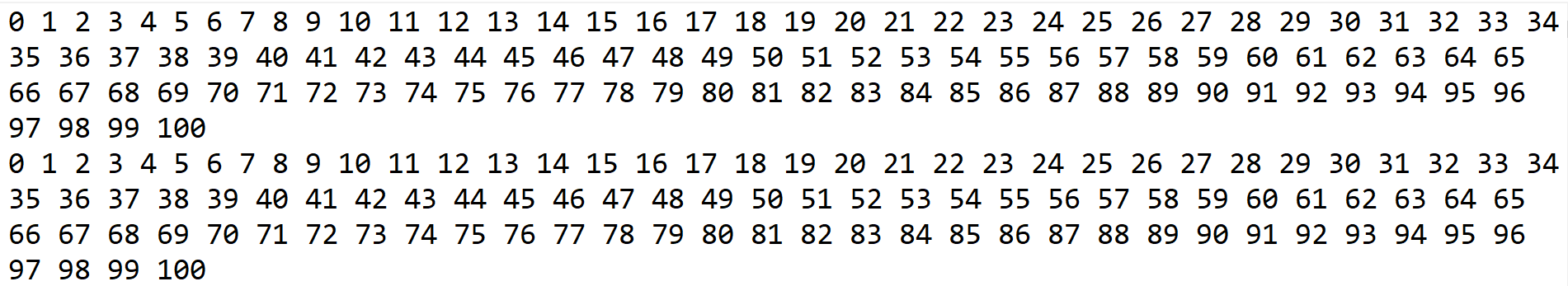


Рисунок 2.3 – Результат записи данных в файл

ЗАКЛЮЧЕНИЕ

В данной лабораторной работе была реализована обработка содержимого файла данных с использованием асинхронного ввода-вывода (I/O). В ходе эксперимента была выполнена параллельная инициированная загрузка данных и их обработка в памяти, что позволило оптимизировать производительность программы по сравнению с традиционным последовательным подходом (чтение → обработка → выгрузка).

Было проведено варьирование количества одновременно выполняемых операций ввода-вывода для анализа влияния параллелизма на общую производительность системы. Результаты показали, что при увеличении числа параллельных операций достигается ускорение процесса обработки данных, однако существует порог, после которого дальнейшее увеличение количества параллельно выполняемых операций не приводит к значительному улучшению производительности. Это связано с наличием "узкого места" в программной реализации, которое проявляется на уровне ограниченной пропускной способности ввода-вывода или из-за особенностей архитектуры системы, таких как ограниченные ресурсы процессора или памяти.

Сравнительная оценка эффективности асинхронного подхода по сравнению с традиционным показала, что асинхронная реализация значительно быстрее в сценариях, где операции ввода-вывода занимают существенную часть времени. Опционально было произведено сравнение с многопоточной реализацией, результаты которого продемонстрировали, что в большинстве случаев асинхронный подход был более производителен за счет лучшего управления ресурсами и меньшего количества затрат на переключение контекстов между потоками.

Таким образом, использование асинхронного ввода-вывода является эффективным методом повышения производительности при обработке больших объемов данных, особенно в случаях, когда операции ввода-вывода занимают продолжительное время.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] CreateFile function [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea.

[2] ReadFile function [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/fileapi/nf-fileapi-readfile.

[3] WriteFile function [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/fileapi/nf-fileapi-writefile.

[4] Overlapped structure [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-overlapped.

ПРИЛОЖЕНИЕ А  
(обязательное)  
Исходный код программы

#include <windows.h>

#include <iostream>

#include <vector>

#include <sstream>

#include <chrono>

#include <fstream>

#include "tools.cpp"

using namespace std;

const int chunk\_sizes[] = {25, 296, 3897};

const DWORD CHUNK\_SIZE = 296;

const int string\_number = 100000;

#define THREADS 5

struct ThreadInfo

{

OVERLAPPED overlapped; // Structure for asynchronous I/O

HANDLE file\_handle; // Handle for the file being read/written

char\* buffer; // Buffer to store data

size\_t size; // Size of the data to read/write

size\_t offset; // Offset for reading/writing

};

DWORD WINAPI ReadFileAsync(LPVOID args)

{

auto start = chrono::steady\_clock::now();

ThreadInfo\* io\_data = (ThreadInfo\*)args; // Cast parameter to AsyncIOData

DWORD bytes\_read;

// Set the offset for the read operation

io\_data->overlapped.Offset = (DWORD)io\_data->offset;

io\_data->overlapped.OffsetHigh = (DWORD)(io\_data->offset >> 32);

// Start the read operation

if (!ReadFile(io\_data->file\_handle, io\_data->buffer, io\_data->size, &bytes\_read, &io\_data->overlapped))

{

// Check if the read operation is pending

if (GetLastError() != ERROR\_IO\_PENDING)

{

perror("Failed to initiate read");

return 1;

}

}

// Wait for the read operation to complete

GetOverlappedResult(io\_data->file\_handle, &io\_data->overlapped, &bytes\_read, TRUE);

auto end = chrono::steady\_clock::now();

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

//io\_data->time = duration.count();

return bytes\_read;

}

void Processing(char\* buffer, int size)

{

vector<string> lines = SplitBufferIntoLines(buffer, size);

vector<int> f;

string line;

size\_t currentPos = 0;

for (int i = 0; i < lines.size(); i++)

{

f = CharArrayToIntVector(lines[i]);

bubblesort(&f, f.size());

line = VectorToString(f);

size\_t lineSize = line.size();

memcpy(buffer + currentPos, line.c\_str(), lineSize);

currentPos += lineSize;

}

}

DWORD WINAPI WriteFileAsync(LPVOID args)

{

ThreadInfo\* io\_data = (ThreadInfo\*)args;

DWORD bytes\_written;

// Set the offset for the write operation

io\_data->overlapped.Offset = (DWORD)io\_data->offset;

io\_data->overlapped.OffsetHigh = (DWORD)(io\_data->offset >> 32);

// Start the write operation

if (!WriteFile(io\_data->file\_handle, io\_data->buffer, io\_data->size, &bytes\_written, &io\_data->overlapped))

{

// Check if the write operation is pending

if (GetLastError() != ERROR\_IO\_PENDING)

{

perror("Failed to initiate write");

return 1;

}

}

// Wait for the write operation to complete

GetOverlappedResult(io\_data->file\_handle, &io\_data->overlapped, &bytes\_written, TRUE);

return bytes\_written; // Return number of bytes written

}

int ProcessFileAsync(const wstring& filenameRead, const wstring& filenameWrite) {

auto start = chrono::steady\_clock::now();

HANDLE hFileR = CreateFile(

filenameRead.c\_str(), // Имя файла

GENERIC\_READ, // Доступ на чтение

0, // Нет совместного доступа

NULL, // Атрибуты безопасности

OPEN\_EXISTING, // Открыть существующий файл

FILE\_FLAG\_OVERLAPPED && FILE\_ATTRIBUTE\_NORMAL, // Асинхронный ввод-вывод

NULL // Шаблон файла

);

HANDLE hFileW = CreateFile(

filenameWrite.c\_str(), // Имя файла

GENERIC\_WRITE, // Доступ на запись

0, // Нет совместного доступа

NULL, // Атрибуты безопасности

OPEN\_ALWAYS, // Открыть существующий или создать новый файл

FILE\_FLAG\_OVERLAPPED && FILE\_ATTRIBUTE\_NORMAL, // Асинхронный ввод-вывод

NULL // Шаблон файла

);

LARGE\_INTEGER fileSize;

GetFileSizeEx(hFileR, &fileSize);

//cout << fileSize.QuadPart << '\n';

SetFilePointerEx(hFileW, fileSize, NULL, FILE\_BEGIN);

SetEndOfFile(hFileW);

// Используем SetFileValidData для установки валидных данных

SetFileValidData(hFileW, fileSize.QuadPart);

//LARGE\_INTEGER fileSize;

//GetFileSizeEx(hFileR, &fileSize);

//SetFilePointerEx(hFileW, fileSize, NULL, FILE\_BEGIN);

//

/\* int s;

cin >> s;\*/

/\*if (hFile == INVALID\_HANDLE\_VALUE) {

cerr << "Не удалось открыть файл, ошибка: " << GetLastError() << endl;

return;

}\*/

OVERLAPPED overlappedR = { 0 };

overlappedR.Offset = 0; // Начало чтения с начала файла

overlappedR.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

OVERLAPPED overlappedW = { 0 };

overlappedW.Offset = 0; // Пишем с начала файла

overlappedW.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

/\*if (overlapped.hEvent == NULL) {

cerr << "Не удалось создать событие, ошибка: " << GetLastError() << endl;

CloseHandle(hFile);

return;

}\*/

char buffer[CHUNK\_SIZE] = { 0 };

DWORD bytesRead = 0;

DWORD bytesWritten = 0;

while (true) {

BOOL readResult = ReadFile(

hFileR,

buffer,

CHUNK\_SIZE,

&bytesRead,

&overlappedR

);

if (!readResult) {

if (GetLastError() == ERROR\_IO\_PENDING) {

// Ожидаем завершения асинхронного чтения

WaitForSingleObject(overlappedR.hEvent, INFINITE);

GetOverlappedResult(hFileR, &overlappedR, &bytesRead, FALSE);

}

else {

cerr << "Ошибка чтения файла, код: " << GetLastError() << endl;

break;

}

}

if (bytesRead == 0) {

// Конец файла

break;

}

//cout << "BUFFER: " << buffer << '\n';

string str(buffer);

str = str.substr(0, CHUNK\_SIZE - 3);

//cout << "BUFFER-2: " << str << '\n';

ZeroMemory(buffer, sizeof(buffer));

vector<int> arr = CharArrayToIntVector(str);

/\*cout << "BEFORE BUBBLE: ";

for (int i = 0; i < arr.size(); i++)

cout << arr[i] << ' ';

cout << endl;\*/

bubblesort(&arr, arr.size());

/\* cout << "AFTER BUBBLE: ";

for (int i = 0; i < arr.size(); i++)

cout << arr[i] << ' ';

cout << endl;\*/

string res = VectorToString(arr);

//cout << "after: " << res;

//arr.clear();

// Обновляем смещение для следующего чтения

overlappedR.Offset += CHUNK\_SIZE;

ResetEvent(overlappedR.hEvent);

BOOL writeResult = WriteFile(

hFileW, // Дескриптор файла

res.c\_str(), // Данные для записи

CHUNK\_SIZE-2, // Размер данных для записи

&bytesWritten, // Фактически записанные байты

&overlappedW // OVERLAPPED структура

);

if (!writeResult) {

if (GetLastError() == ERROR\_IO\_PENDING) {

// Ожидаем завершения асинхронной записи

WaitForSingleObject(overlappedW.hEvent, INFINITE);

// Получаем результат асинхронной записи

if (!GetOverlappedResult(hFileW, &overlappedW, &bytesWritten, FALSE)) {

/\*cerr << "Ошибка записи в файл, код: " << GetLastError() << endl;\*/

break;

}

}

else {

/\*cerr << "Ошибка записи в файл, код: " << GetLastError() << endl;\*/

break;

}

}

overlappedW.Offset += bytesWritten;

ResetEvent(overlappedW.hEvent);

}

// Закрытие дескрипторов

CloseHandle(overlappedW.hEvent);

CloseHandle(hFileW);

CloseHandle(overlappedR.hEvent);

CloseHandle(hFileR);

auto end = chrono::steady\_clock::now();

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

cout << "Async DONE : " << duration.count() << '\n';

return duration.count();

}

int ProcessFile(const wstring& filenameRead, const wstring& filenameWrite) {

auto start = chrono::steady\_clock::now();

HANDLE hFileR = CreateFile(

filenameRead.c\_str(),

GENERIC\_READ,

FILE\_SHARE\_READ,

NULL,

OPEN\_EXISTING,

FILE\_ATTRIBUTE\_NORMAL,

NULL

);

HANDLE hFileW = CreateFile(

filenameWrite.c\_str(), // Имя файла

GENERIC\_WRITE, // Доступ на запись

0, // Нет совместного доступа

NULL, // Атрибуты безопасности

CREATE\_ALWAYS, // Создать файл, если не существует, или перезаписать, если существует

FILE\_ATTRIBUTE\_NORMAL, // Обычный файл

NULL // Шаблон файла

);

if (hFileR == INVALID\_HANDLE\_VALUE) {

cerr << "Ошибка открытия файла: " << GetLastError() << endl;

return 0;

}

const DWORD bufferSize = CHUNK\_SIZE; // Размер буфера для чтения

char buffer[bufferSize];

DWORD bytesRead = 0;

DWORD bytesWritten = 0;

while (true) {

BOOL result = ReadFile(hFileR, buffer, bufferSize, &bytesRead, NULL);

if (!result || bytesRead == 0) {

break; // Выход из цикла, если произошла ошибка или достигнут конец файла

}

//cout << "BUFFER: " << buffer << '\n';

string str(buffer);

str = str.substr(0, CHUNK\_SIZE - 3);

//cout << "BUFFER-2: " << str << '\n';

ZeroMemory(buffer, sizeof(buffer));

vector<int> arr = CharArrayToIntVector(str);

/\*cout << "BEFORE BUBBLE: ";

for (int i = 0; i < arr.size(); i++)

cout << arr[i] << ' ';

cout << endl;

\*/

bubblesort(&arr, arr.size());

/\*cout << "AFTER BUBBLE: ";

for (int i = 0; i < arr.size(); i++)

cout << arr[i] << ' ';

cout << endl;\*/

//cout << arr.size() << "ttt";

string res = VectorToString(arr);

/\*res += buffer[22];

res += buffer[23];

res += buffer[24];

cout << res.size() << "yyy";\*/

BOOL writeResult = WriteFile(

hFileW, // Дескриптор файла

res.c\_str(), // Данные для записи

CHUNK\_SIZE -2, // Размер данных для записи

&bytesWritten, // Фактически записанные байты

NULL // OVERLAPPED не используется для синхронной записи

);

if (!writeResult || bytesWritten == 0) {

break;

}

}

CloseHandle(hFileR); // Закрываем дескриптор файла

CloseHandle(hFileW);

auto end = chrono::steady\_clock::now();

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

//cout << "DONE : " << duration.count() << '\n';

return duration.count();

}

char\* ReadFromFile(const wstring& filename) {

// Открываем файл для чтения

HANDLE hFile = CreateFile(

filename.c\_str(), // Имя файла

GENERIC\_READ, // Доступ на чтение

0, // Нет совместного доступа

NULL, // Атрибуты безопасности

OPEN\_EXISTING, // Открыть существующий файл

FILE\_ATTRIBUTE\_NORMAL, // Обычный файл

NULL // Шаблон файла

);

if (hFile == INVALID\_HANDLE\_VALUE) {

cerr << "Не удалось открыть файл, ошибка: " << GetLastError() << endl;

return nullptr;

}

// Получаем размер файла

LARGE\_INTEGER fileSize;

if (!GetFileSizeEx(hFile, &fileSize)) {

cerr << "Не удалось получить размер файла, ошибка: " << GetLastError() << endl;

CloseHandle(hFile);

return nullptr;

}

// Проверяем, не превышает ли размер файл допустимый диапазон для чтения в память

if (fileSize.QuadPart > MAXDWORD) {

cerr << "Файл слишком большой для чтения целиком!" << endl;

CloseHandle(hFile);

return nullptr;

}

// Буфер для чтения данных из файла

DWORD bytesRead;

char\* buffer = new char[static\_cast<DWORD>(fileSize.QuadPart) + 1]; // Плюс 1 для завершающего нуля

// Чтение данных из файла

if (!ReadFile(hFile, buffer, static\_cast<DWORD>(fileSize.QuadPart), &bytesRead, NULL))

{

cerr << "Ошибка чтения файла, код: " << GetLastError() << endl;

delete[] buffer;

CloseHandle(hFile);

return nullptr;

}

// Завершаем строку нулевым символом

buffer[bytesRead] = '\0';

// Выводим данные на экран

//cout << "Содержимое файла:\n" << buffer << endl;

CloseHandle(hFile);

return buffer;

}

string ProcessData(const char\* charPtr, size\_t n) {

// Приводим указатель char\* к string

string str(charPtr);

vector<string> substrings;

// Цикл, который проходит по строке и разбивает её на части длиной n

for (size\_t i = 0; i < str.size(); i += n) {

substrings.push\_back(str.substr(i, n));

}

string processedData= "";

for (size\_t i = 0; i < substrings.size(); i ++) {

vector<int> arr = CharArrayToIntVector(substrings[i]);

/\*for (int j = 0; j < arr.size(); j++)

cout << arr[j] << " ";

cout << endl;\*/

bubblesort(&arr, arr.size());

string res = VectorToString(arr);

// cout << "res " << res;

processedData += res;

}

// cout << "processedData " << processedData<<"\n";

return processedData;

}

bool WriteToFile(const wstring& filename, string& data) {

// Открываем файл для записи, создаем новый если он не существует

HANDLE hFile = CreateFile(

filename.c\_str(), // Имя файла

GENERIC\_WRITE, // Доступ на запись

0, // Нет совместного доступа

NULL, // Атрибуты безопасности

CREATE\_ALWAYS, // Создать файл, если не существует, или перезаписать, если существует

FILE\_ATTRIBUTE\_NORMAL, // Обычный файл

NULL // Шаблон файла

);

if (hFile == INVALID\_HANDLE\_VALUE) {

cerr << "Не удалось открыть файл, ошибка: " << GetLastError() << endl;

return false;

}

DWORD bytesWritten = 0;

// Записываем данные в файл

BOOL writeResult = WriteFile(

hFile, // Дескриптор файла

data.c\_str(), // Данные для записи

static\_cast<DWORD>(data.size()), // Размер данных для записи

&bytesWritten, // Фактически записанные байты

NULL // OVERLAPPED не используется для синхронной записи

);

if (!writeResult) {

cerr << "Ошибка записи в файл, код: " << GetLastError() << endl;

CloseHandle(hFile);

return false;

}

//cout << "Записано байт: " << bytesWritten << endl;

// Закрываем файл

CloseHandle(hFile);

return true;

}

int ProcessFile2(const wstring& filenameRead, const wstring& filenameWrite)

{

auto start = chrono::steady\_clock::now();

char\* buffer = ReadFromFile(filenameRead);

string data = ProcessData(buffer, CHUNK\_SIZE);

WriteToFile(filenameWrite, data);

auto end = chrono::steady\_clock::now();

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

//cout << "DONE : " << duration.count() << '\n';

return duration.count();

}

void writeSequenceInReverseToFile(const wstring& filename, const vector<int>& sequence)

{

ofstream outFile(filename);

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

cerr << "Ошибка открытия файла: " << filename.c\_str() << endl;

return;

}

for (int i = 0; i < string\_number; i++)

{

// Записываем в файл

for (const int& number : sequence) {

outFile << number << ' ';

}

outFile << '\n';

}

outFile.close(); // Закрываем файл

}

int main() {

wstring fileNameR = L"start.txt";

wstring fileNameW = L"end.txt";

wstring fileNameR1 = L"start1.txt";

wstring fileNameW1 = L"end1.txt";

vector<int> a;

for (int i = 100; i >= 0; i--)

{

a.push\_back(i);

}

writeSequenceInReverseToFile(fileNameR, a);

writeSequenceInReverseToFile(fileNameR1, a);

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

HANDLE fileReadHandle = CreateFile(fileNameR.c\_str(), GENERIC\_READ, FILE\_SHARE\_READ, NULL, OPEN\_EXISTING, FILE\_FLAG\_OVERLAPPED, NULL);

if (fileReadHandle == INVALID\_HANDLE\_VALUE)

{

perror("Failed to open input file");

return EXIT\_FAILURE;

}

LARGE\_INTEGER file\_size;

// Get the size of the input file

if (!GetFileSizeEx(fileReadHandle, &file\_size))

{

CloseHandle(fileReadHandle);

perror("Failed to get file size");

return EXIT\_FAILURE;

}

cout<<"File size: B "<<file\_size.QuadPart<<'\n';

size\_t chunk\_size = file\_size.QuadPart / THREADS; // Calculate chunk size for each thread

HANDLE read\_threads[THREADS]; // Array to hold read thread handles

HANDLE write\_threads[THREADS]; // Array to hold write thread handles

ThreadInfo read\_data[THREADS]; // Array to hold read data structures

ThreadInfo write\_data[THREADS]; // Array to hold write data structures

char\* buffers[THREADS]; // Array of buffers for each thread

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

{

buffers[i] = (char\*)malloc(chunk\_size); // Allocate memory for buffer

if (!buffers[i])

{

perror("Failed to allocate memory for buffer");

return EXIT\_FAILURE;

}

// Initialize read data for each thread

read\_data[i].overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

read\_data[i].file\_handle = fileReadHandle;

read\_data[i].buffer = buffers[i];

read\_data[i].size = chunk\_size;

read\_data[i].offset = i \* chunk\_size;

// Create a thread for reading

read\_threads[i] = CreateThread(NULL, 0, ReadFileAsync, &read\_data[i], 0, NULL);

if (!read\_threads[i])

{

perror("Failed to create read thread");

return EXIT\_FAILURE;

}

}

// Wait for all read operations to complete

WaitForMultipleObjects(THREADS, read\_threads, TRUE, INFINITE);

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

{

CloseHandle(read\_threads[i]); // Close thread handles

}

/\*for (int i = 0; i < THREADS; ++i)

{

cout<< i << ") ";

for (int j = 0; j < read\_data[i].size; ++j)

{

cout<< read\_data[i].buffer[j];

}

cout << '\n';

}\*/

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

{

Processing(read\_data[i].buffer, read\_data[i].size);

}

//cout << read\_data[0].size << '\n';

/\*for (int i = 0; i < THREADS; ++i)

{

cout << i << ") ";

for (int j = 0; j < read\_data[i].size - string\_number \* 2 / THREADS; ++j)

{

cout << read\_data[i].buffer[j];

}

cout << '\n';

}\*/

// Create output file handle for writing

HANDLE fileWriteHandle = CreateFile(fileNameW.c\_str(), GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_FLAG\_OVERLAPPED, NULL);

if (fileWriteHandle == INVALID\_HANDLE\_VALUE)

{

CloseHandle(fileReadHandle);

perror("Failed to open output file");

return EXIT\_FAILURE;

}

// Write operations in parallel

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

{

// Initialize write data for each thread

write\_data[i].overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

write\_data[i].file\_handle = fileWriteHandle;

write\_data[i].buffer = buffers[i];

write\_data[i].size = chunk\_size - string\_number\*2/THREADS;

write\_data[i].offset = i \* (chunk\_size - string\_number \* 2 / THREADS);

// Create a thread for writing

write\_threads[i] = CreateThread(NULL, 0, WriteFileAsync, &write\_data[i], 0, NULL);

if (!write\_threads[i])

{

perror("Failed to create write thread");

return EXIT\_FAILURE;

}

}

// Wait for all write operations to complete

WaitForMultipleObjects(THREADS, write\_threads, TRUE, INFINITE);

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

{

CloseHandle(write\_threads[i]); // Close thread handles

free(buffers[i]); // Free allocated buffers

}

CloseHandle(fileReadHandle);

CloseHandle(fileWriteHandle);

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

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

int async = duration\_ms.count();

int sync2 = ProcessFile2(fileNameR1, fileNameW1);

//int sync = ProcessFile(fileNameR1, fileNameW1);

cout << "Async - " << async << '\n';

//cout << "Sync - " << sync << '\n';

cout << "Sync - " << sync2 << '\n';

cout << double (sync2 - async) / async \* 100.;

return 0;

}