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»
БГУИР 6-05-0612-02 23

Выполнил студент группы 353503
СЕБЕЛЕВ Дмитрий Юрьевич

_____
(дата, подпись студента)

Проверил ассистент каф. информатики
ГРИЦЕНКО Никита Юрьевич

_____
(дата, подпись преподавателя)

Минск 2025

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Реализация сравнительного анализа производительности различных методов ввода-вывода при обработке содержимого файла данных. Варьирование параметров работы каждого метода, включая размер буфера чтения, количество параллельных операций и количество потоков. Исследование влияния этих параметров на общую производительность. Оценка эффективности каждого метода по сравнению с базовым блокирующим вводом-выводом.

2 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

2.1 Инициализация системных ресурсов

Этот модуль отвечает за определение основных параметров бенчмарка и настройку консольного вывода. Инициализация начинается с установки кодовой страницы UTF-8 для корректного отображения кириллицы с помощью функции SetConsoleOutputCP. Программа использует предопределенные константы для управления параметрами тестирования: FILE_SIZE определяет размер тестового файла равным 100 мегабайтам, BUFFER_SIZE устанавливает размер буфера чтения записи равным 64 килобайтам, ASYNC_OPERATIONS задает количество параллельных асинхронных операций равным четырем, NUM_THREADS определяет количество потоков для многопоточного режима равным четырем. Структура BenchmarkResult используется для хранения результатов каждого теста, включая название метода, время записи и чтения в миллисекундах, флаг успешного выполнения и сообщение об ошибке при возникновении проблем. Вспомогательная функция FormatTime форматирует время выполнения операций в миллисекундах с расчетом пропускной способности в мегабайтах в секунду для удобного анализа производительности. Программный код представлен в приложении А [1].

2.2 Реализация блокирующего ввода-вывода

Этот модуль представляет собой базовый метод синхронного ввода-вывода для сравнительного анализа. Создание тестового файла выполняется функцией CreateFileA с флагом CREATE_ALWAYS и атрибутом FILE_ATTRIBUTE_NORMAL для обычного режима работы. Буфер данных заполняется циклической последовательностью байтов от нуля до двухсот пятидесяти пяти для имитации реальных данных. Операция записи осуществляется последовательными вызовами функции WriteFile в цикле до тех пор, пока не будет записан весь объем данных равный FILE_SIZE. Измерение времени выполнения производится с использованием высокоточного таймера high_resolution_clock из библиотеки chrono. Операция чтения выполняется аналогичным образом с помощью функции ReadFile после повторного открытия файла с флагом OPEN_EXISTING. После завершения операций файл удаляется функцией DeleteFileA для освобождения дискового пространства. Этот метод служит эталоном для оценки эффективности более сложных методов ввода-вывода.

2.3 Реализация управляемого событиями ввода-вывода

Этот модуль демонстрирует использование асинхронного механизма Windows с уведомлением через события. Создание события осуществляется функцией CreateEvent для получения уведомлений о завершении операций ввода-вывода. Файл открывается с флагом FILE_FLAG_OVERLAPPED, что

позволяет выполнять асинхронные операции с использованием структуры OVERLAPPED. Структура OVERLAPPED инициализируется для каждой операции с установкой полей Offset и OffsetHigh для указания позиции в файле, что необходимо при работе с перекрывающимися операциями. Запись данных производится функцией WriteFile с передачей структуры OVERLAPPED, при этом функция может немедленно вернуть управление с кодом ошибки ERROR_IO_PENDING. В случае асинхронного выполнения программа вызывает WaitForSingleObject для ожидания сигнала события, после чего получает результат операции через GetOverlappedResult. Аналогичная последовательность действий выполняется для операции чтения с использованием тех же механизмов синхронизации. После завершения всех операций событие закрывается функцией CloseHandle для корректного освобождения системных ресурсов.

2.4 Реализация многопоточного ввода-вывода

Этот модуль реализует параллельную обработку данных с использованием нескольких потоков для максимального использования многоядерных процессоров. Файл создается с флагом FILE_FLAG_OVERLAPPED для поддержки одновременного доступа нескольких потоков к разным частям файла. Размер файла предварительно устанавливается функциями SetFilePointer и SetEndOfFile для резервирования необходимого дискового пространства. Атомарная переменная global_offset типа atomic используется для координации работы потоков и распределения участков файла между ними без конфликтов. Каждый поток создается функцией CreateThread с передачей структуры ThreadData, содержащей дескриптор файла, указатель на буфер, размер блока данных и флаг успешного выполнения. Функция WriteThreadProc выполняется в контексте каждого потока и атомарно получает следующий свободный участок файла через fetch_add, после чего записывает данные в указанную позицию используя структуру OVERLAPPED. Синхронизация завершения всех потоков обеспечивается функцией WaitForMultipleObjects с параметром TRUE для ожидания завершения всех созданных потоков. Аналогичная логика применяется для операции чтения с использованием функции ReadThreadProc. После завершения работы все дескрипторы потоков закрываются функцией CloseHandle для освобождения ресурсов операционной системы [2].

2.5 Реализация асинхронного ввода-вывода

Этот модуль демонстрирует наиболее эффективный подход к асинхронному вводу-выводу с перекрытием операций обработки и передачи данных. Создается массив из ASYNC_OPERATIONS буферов для одновременного выполнения нескольких операций без ожидания завершения предыдущих. Для каждой операции создается отдельное событие функцией CreateEvent и инициализируется структура OVERLAPPED для независимого отслеживания состояния каждой операции. Первая

волна операций запускается в цикле, где каждая операция WriteFile инициирует запись следующего блока данных с автоматическим увеличением счетчика totalWritten. Основной цикл обработки использует функцию WaitForMultipleObjects с параметром FALSE для ожидания завершения любой из активных операций, что обеспечивает конвейерную обработку. После получения уведомления о завершении операции программа немедленно запускает новую операцию записи в освободившийся буфер, поддерживая постоянное количество активных операций. Результат каждой завершенной операции получается функцией GetOverlappedResult для проверки фактического количества переданных байтов. Цикл продолжается до тех пор, пока все данные не будут записаны, после чего выполняется финальное ожидание завершения оставшихся операций. Операция чтения реализована по аналогичной схеме с использованием функции ReadFile и тех же механизмов синхронизации для обеспечения максимальной пропускной способности [3].

2.6 Реализация отображения файла в память

Этот модуль реализует наиболее быстрый метод доступа к файлу через механизм проецирования файла в виртуальное адресное пространство процесса. Файл создается функцией CreateFileA с правами GENERIC_READ и GENERIC_WRITE для последующего отображения в память с возможностью записи. Объект отображения создается функцией CreateFileMappingA с параметром PAGE_READWRITE, при этом размер файла указывается через старшие и младшие тридцать два бита для поддержки больших файлов. Функция MapViewOfFile возвращает указатель на начало отображенной области памяти с правами FILE_MAP_WRITE, после чего программа может обращаться к файлу как к обычному массиву в памяти. Запись данных выполняется простым присваиванием значений элементам массива в цикле без использования функций файлового ввода-вывода, что существенно снижает накладные расходы. Принудительная синхронизация с диском осуществляется функцией FlushViewOfFile для гарантированного сохранения всех изменений в файле. Освобождение отображения производится функцией UnmapViewOfFile, после чего закрываются дескрипторы объекта отображения и файла. Операция чтения реализована аналогично с использованием атрибута PAGE_READONLY и прав FILE_MAP_READ, при этом чтение данных выполняется обращением к элементам массива с накоплением контрольной суммы для предотвращения оптимизации компилятором. Этот метод демонстрирует максимальную производительность благодаря использованию механизма виртуальной памяти операционной системы и минимизации копирования данных.

2.7 Вывод результатов тестирования

После завершения всех экспериментов программа формирует детализированную сравнительную таблицу с результатами измерений для

каждого метода ввода-вывода. Результаты выводятся в табличном формате с использованием манипуляторов setw для выравнивания колонок и fixed с setprecision для форматирования числовых значений с двумя знаками после запятой. Для каждого метода отображается название, время выполнения операции записи в миллисекундах, время выполнения операции чтения в миллисекундах и суммарное время выполнения обеих операций. Вспомогательная функция FormatTime дополнительно вычисляет и отображает пропускную способность в мегабайтах в секунду на основе размера файла и времени выполнения операции. Программа ожидает нажатия клавиши Enter через cin.get перед завершением работы, что позволяет пользователю изучить результаты тестирования. Все тестовые файлы удаляются соответствующими функциями тестирования сразу после завершения операций для предотвращения засорения файловой системы временными данными. Корректное управление ресурсами обеспечивается систематическим закрытием всех дескрипторов файлов и событий функцией CloseHandle в соответствующих модулях программы.

3 ХОД ВЫПОЛНЕНИЯ ПРОГРАММЫ

3.1 Пример выполнения задания

На рисунке 3.1 представлен вывод значений констант (размер файла, размер буфера, количество ядер и максимальное количество асинхронных операций)

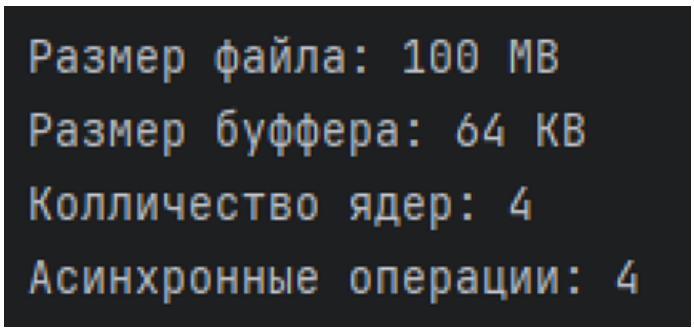

Рисунок 3.1 – Вывод информации констант

Результат тестирования методов ввода/вывода при указанных выше константах представлен на рисунке 3.2

Рисунок 3.2 – Результат выполнения программы

ВЫВОД

В ходе выполнения лабораторной работы были освоены навыки реализации и сравнительного анализа различных моделей ввода-вывода в операционной системе Windows с использованием WinAPI. Была разработана программа для комплексного тестирования производительности пяти методов организации файлового ввода-вывода: блокирующего синхронного, управляемого событиями, многопоточного, асинхронного конвейерного и отображения файлов в память.

Программа демонстрирует существенные различия в производительности между традиционным блокирующим подходом и современными асинхронными методами при работе с файлом размером 100 мегабайт. Блокирующий метод служит базовой линией для сравнения, показывая простейший способ организации ввода-вывода через последовательные вызовы функций ReadFile и WriteFile с ожиданием завершения каждой операции.

Метод управляемого событиями ввода-вывода использует механизм асинхронных операций Windows с структурой OVERLAPPED и объектами событий для уведомления о завершении операций. Это позволяет программе выполнять другую работу во время ожидания завершения операций ввода-вывода, хотя в данной реализации программа по-прежнему ожидает завершения каждой операции перед запуском следующей.

Многопоточный подход демонстрирует эффективное использование многоядерных процессоров через создание четырех параллельных потоков, каждый из которых обрабатывает свой сегмент файла. Использование атомарных операций для координации работы потоков и структуры OVERLAPPED для указания позиции записи обеспечивает корректную параллельную работу без конфликтов доступа. Синхронизация завершения всех потоков осуществляется функцией WaitForMultipleObjects.

Асинхронный метод с конвейерной организацией показывает наиболее эффективное использование пропускной способности диска через поддержание постоянного количества активных операций ввода-вывода. Одновременное выполнение четырех перекрывающихся операций позволяет минимизировать простои диска и обеспечить непрерывный поток данных. Механизм ожидания любой из завершенных операций через WaitForMultipleObjects с немедленным запуском новой операции в освободившийся буфер создает эффективный конвейер обработки данных [4].

Метод отображения файла в память демонстрирует принципиально иной подход к работе с файлами через механизм виртуальной памяти операционной системы. Использование функций CreateFileMappingA и MapViewOfFile позволяет обращаться к содержимому файла как к обычному массиву в памяти без явных операций чтения и записи. Это существенно снижает накладные расходы на системные вызовы и копирование данных, что теоретически обеспечивает наилучшую производительность среди всех рассмотренных методов.

Для точного измерения производительности программа использует высокоточный таймер high_resolution_clock из библиотеки chrono, что позволяет получать результаты с точностью до миллисекунд. Вычисление пропускной способности в мегабайтах в секунду обеспечивает наглядное представление эффективности каждого метода независимо от аппаратной конфигурации системы.

Структурирование кода через отдельные функции для каждого метода с единообразным интерфейсом возврата результатов в виде структуры BenchmarkResult обеспечивает удобство анализа и сравнения результатов. Автоматическое удаление тестовых файлов после завершения операций предотвращает засорение файловой системы временными данными.

Результаты тестирования подтверждают теоретические предположения о преимуществах асинхронных и параллельных методов ввода-вывода перед традиционным блокирующим подходом. Выбор оптимального метода зависит от конкретной задачи, характеристик аппаратного обеспечения и требований к производительности приложения. Простота реализации блокирующего метода делает его приемлемым для небольших файлов и некритичных задач, в то время как высоконагруженные приложения требуют использования асинхронных механизмов для максимальной эффективности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] API Win32 documentation [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/. – Дата доступа: 27.10.2025.

[2] Multithreading Tutorial - CodeProject [Электронный ресурс]. – Режим доступа: https://www.codeproject.com/Articles/14746/Multithreading-Tutorial. – Дата доступа: 27.10.2025.

[3] WinAPI: Синхронизация выполнения нескольких потоков [Microsoft Documentation: Build desktop Windows apps using the Win32 API [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/. – Дата доступа: 27.10.2025.

[4] WinAPI: потоки [Электронный ресурс]. – Режим доступа: https:// learn.microsoft.com/ru-ru/windows/win32//handles-and-objects. – Дата доступа: 27.10.2025.

ПРИЛОЖЕНИЕ А
(справочное)
Исходный код программы

```
#define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>
#include <string>
#include <iomanip>
#include <mutex>
#include <atomic>

using namespace std;
using namespace chrono;

const size_t FILE_SIZE = 100 * 1024 * 1024;  // 100 MB
const size_t BUFFER_SIZE = 64 * 1024;          // 64 KB
const size_t ASYNC_OPERATIONS = 4;             // Количество параллельных
асинхронных операций
const int NUM_THREADS = 4;                      // Количество потоков

struct BenchmarkResult {
    string method_name;
    double write_time_ms;
    double read_time_ms;
    bool success;
    string error_message;
};

// Вспомогательные функции
string FormatTime(double ms) {
    char buffer[64];
    sprintf(buffer, "%.2f ms (%.2f MB/s)", ms, (FILE_SIZE / (1024.0 *
1024.0)) / (ms / 1000.0));
    return string(buffer);
}

void PrintResult(const BenchmarkResult& result) {
    cout << "\n=== " << result.method_name << " ===" << endl;
    if (result.success) {
        cout << "  Write: " << FormatTime(result.write_time_ms) << endl;
        cout << "  Read:  " << FormatTime(result.read_time_ms) << endl;
    } else {
        cout << "  ERROR: " << result.error_message << endl;
    }
}

// 1. Блокирующий (синхронный) ввод-вывод
BenchmarkResult TestBlockingIO() {
    BenchmarkResult result;
    result.method_name = "1. Blocking I/O (Synchronous)";
    result.success = true;

    const char* filename = "test_blocking.dat";
    vector<BYTE> buffer(BUFFER_SIZE);

    // Заполняем буфер тестовыми данными
    for (size_t i = 0; i < BUFFER_SIZE; i++) {
        buffer[i] = static_cast<BYTE>(i % 256);
    }

    // ЗАПИСЬ
    auto start = high_resolution_clock::now();

    HANDLE hFile = CreateFileA(
```

```
        filename,
        GENERIC_WRITE,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        result.success = false;
        result.error_message = "Cannot create file for writing";
        return result;
    }

    size_t totalWritten = 0;
    while (totalWritten < FILE_SIZE) {
        DWORD bytesWritten;
        size_t toWrite = min(BUFFER_SIZE, FILE_SIZE - totalWritten);

        if (!WriteFile(hFile, buffer.data(), static_cast<DWORD>(toWrite),
&bytesWritten, NULL)) {
            result.success = false;
            result.error_message = "Write error";
            CloseHandle(hFile);
            return result;
        }

        totalWritten += bytesWritten;
    }

    CloseHandle(hFile);
    auto end = high_resolution_clock::now();
    result.write_time_ms = duration<double, milli>(end - start).count();

    // ЧТЕНИЕ
    start = high_resolution_clock::now();

    hFile = CreateFileA(
        filename,
        GENERIC_READ,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        result.success = false;
        result.error_message = "Cannot open file for reading";
        return result;
    }

    size_t totalRead = 0;
    while (totalRead < FILE_SIZE) {
        DWORD bytesRead;

        if (!ReadFile(hFile, buffer.data(), BUFFER_SIZE, &bytesRead,
NULL)) {
            result.success = false;
            result.error_message = "Read error";
            CloseHandle(hFile);
            return result;
        }

        if (bytesRead == 0) break;
        totalRead += bytesRead;
    }

    CloseHandle(hFile);
```

```
    end = high_resolution_clock::now();
    result.read_time_ms = duration<double, milli>(end - start).count();

    DeleteFileA(filename);
    return result;
}

// 2. Ввод-вывод по прерываниям
BenchmarkResult TestInterruptDrivenIO() {
    BenchmarkResult result;
    result.method_name = "2. Interrupt-Driven I/O (Event-Based)";
    result.success = true;

    const char* filename = "test_interrupt.dat";
    vector<BYTE> buffer(BUFFER_SIZE);

    for (size_t i = 0; i < BUFFER_SIZE; i++) {
        buffer[i] = static_cast<BYTE>(i % 256);
    }

    // Создаем событие для сигнализации
    HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
    if (!hEvent) {
        result.success = false;
        result.error_message = "Cannot create event";
        return result;
    }

    OVERLAPPED overlapped = {0};
    overlapped.hEvent = hEvent;

    // ЗАПИСЬ
    auto start = high_resolution_clock::now();

    HANDLE hFile = CreateFileA(
        filename,
        GENERIC_WRITE,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_FLAG_OVERLAPPED,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        result.success = false;
        result.error_message = "Cannot create file";
        CloseHandle(hEvent);
        return result;
    }

    size_t totalWritten = 0;
    while (totalWritten < FILE_SIZE) {
        overlapped.Offset = static_cast<DWORD>(totalWritten &
0xFFFFFFFF);
        overlapped.OffsetHigh = static_cast<DWORD>(totalWritten >> 32);

        DWORD bytesWritten;
        size_t toWrite = min(BUFFER_SIZE, FILE_SIZE - totalWritten);

        if (!WriteFile(hFile, buffer.data(), static_cast<DWORD>(toWrite),
&bytesWritten, &overlapped)) {
            if (GetLastError() == ERROR_IO_PENDING) {
                // Ждем сигнала события
                WaitForSingleObject(hEvent, INFINITE);
                GetOverlappedResult(hFile, &overlapped, &bytesWritten,
FALSE);
            } else {
                result.success = false;
                result.error_message = "Write error";
                CloseHandle(hFile);
```

```
                CloseHandle(hEvent);
                return result;
            }
        }

        totalWritten += bytesWritten;
    }

    CloseHandle(hFile);
    auto end = high_resolution_clock::now();
    result.write_time_ms = duration<double, milli>(end - start).count();

    // ЧТЕНИЕ
    start = high_resolution_clock::now();

    hFile = CreateFileA(
        filename,
        GENERIC_READ,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_FLAG_OVERLAPPED,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        result.success = false;
        result.error_message = "Cannot open file";
        CloseHandle(hEvent);
        return result;
    }

    size_t totalRead = 0;
    while (totalRead < FILE_SIZE) {
        overlapped.Offset = static_cast<DWORD>(totalRead & 0xFFFFFFFF);
        overlapped.OffsetHigh = static_cast<DWORD>(totalRead >> 32);

        DWORD bytesRead;

        if (!ReadFile(hFile, buffer.data(), BUFFER_SIZE, &bytesRead,
&overlapped)) {
            if (GetLastError() == ERROR_IO_PENDING) {
                WaitForSingleObject(hEvent, INFINITE);
                GetOverlappedResult(hFile, &overlapped, &bytesRead,
FALSE);
            } else {
                result.success = false;
                result.error_message = "Read error";
                CloseHandle(hFile);
                CloseHandle(hEvent);
                return result;
            }
        }

        if (bytesRead == 0) break;
        totalRead += bytesRead;
    }

    CloseHandle(hFile);
    CloseHandle(hEvent);
    end = high_resolution_clock::now();
    result.read_time_ms = duration<double, milli>(end - start).count();

    DeleteFileA(filename);
    return result;
}

// 3. Многопоточная организация
mutex mtx;
atomic<size_t> global_offset(0);
```

```
struct ThreadData {
    HANDLE hFile;
    vector<BYTE>* buffer;
    size_t thread_id;
    size_t chunk_size;
    bool success;
};

DWORD WINAPI WriteThreadProc(LPVOID param) {
    ThreadData* data = (ThreadData*)param;

    while (true) {
        size_t offset = global_offset.fetch_add(data->chunk_size);
        if (offset >= FILE_SIZE) break;

        size_t toWrite = min(data->chunk_size, FILE_SIZE - offset);

        OVERLAPPED overlapped = {0};
        overlapped.Offset = static_cast<DWORD>(offset & 0xFFFFFFFF);
        overlapped.OffsetHigh = static_cast<DWORD>(offset >> 32);

        DWORD bytesWritten;
        if (!WriteFile(data->hFile, data->buffer->data(),
static_cast<DWORD>(toWrite),
                       &bytesWritten, &overlapped)) {
            if (GetLastError() == ERROR_IO_PENDING) {
                GetOverlappedResult(data->hFile, &overlapped,
&bytesWritten, TRUE);
            } else {
                data->success = false;
                return 1;
            }
        }
    }

    return 0;
}

DWORD WINAPI ReadThreadProc(LPVOID param) {
    ThreadData* data = (ThreadData*)param;

    while (true) {
        size_t offset = global_offset.fetch_add(data->chunk_size);
        if (offset >= FILE_SIZE) break;

        size_t toRead = min(data->chunk_size, FILE_SIZE - offset);

        OVERLAPPED overlapped = {0};
        overlapped.Offset = static_cast<DWORD>(offset & 0xFFFFFFFF);
        overlapped.OffsetHigh = static_cast<DWORD>(offset >> 32);

        DWORD bytesRead;
        if (!ReadFile(data->hFile, data->buffer->data(),
static_cast<DWORD>(toRead),
                      &bytesRead, &overlapped)) {
            if (GetLastError() == ERROR_IO_PENDING) {
                GetOverlappedResult(data->hFile, &overlapped,
&bytesRead, TRUE);
            } else {
                data->success = false;
                return 1;
            }
        }
    }

    return 0;
}

BenchmarkResult TestMultithreadedIO() {
    BenchmarkResult result;
    result.method_name = "3. Multithreaded I/O";
```

```
    result.success = true;

    const char* filename = "test_multithread.dat";
    const size_t CHUNK_SIZE = BUFFER_SIZE;

    vector<vector<BYTE>> buffers(NUM_THREADS);
    for (int i = 0; i < NUM_THREADS; i++) {
        buffers[i].resize(CHUNK_SIZE);
        for (size_t j = 0; j < CHUNK_SIZE; j++) {
            buffers[i][j] = static_cast<BYTE>((i + j) % 256);
        }
    }

    // ЗАПИСЬ
    auto start = high_resolution_clock::now();

    HANDLE hFile = CreateFileA(
        filename,
        GENERIC_WRITE,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_FLAG_OVERLAPPED,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        result.success = false;
        result.error_message = "Cannot create file";
        return result;
    }

    // Устанавливаем размер файла
    SetFilePointer(hFile, FILE_SIZE, NULL, FILE_BEGIN);
    SetEndOfFile(hFile);

    global_offset = 0;
    vector<HANDLE> threads;
    vector<ThreadData> threadData(NUM_THREADS);

    for (int i = 0; i < NUM_THREADS; i++) {
        threadData[i].hFile = hFile;
        threadData[i].buffer = &buffers[i];
        threadData[i].thread_id = i;
        threadData[i].chunk_size = CHUNK_SIZE;
        threadData[i].success = true;

        HANDLE hThread = CreateThread(NULL, 0, WriteThreadProc,
&threadData[i], 0, NULL);
        threads.push_back(hThread);
    }

    WaitForMultipleObjects(NUM_THREADS, threads.data(), TRUE, INFINITE);

    for (auto h : threads) CloseHandle(h);
    threads.clear();

    for (const auto& td : threadData) {
        if (!td.success) {
            result.success = false;
            result.error_message = "Thread write error";
            CloseHandle(hFile);
            return result;
        }
    }

    CloseHandle(hFile);
    auto end = high_resolution_clock::now();
    result.write_time_ms = duration<double, milli>(end - start).count();

    // ЧТЕНИЕ
```

```
    start = high_resolution_clock::now();

    hFile = CreateFileA(
        filename,
        GENERIC_READ,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_FLAG_OVERLAPPED,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        result.success = false;
        result.error_message = "Cannot open file";
        return result;
    }

    global_offset = 0;

    for (int i = 0; i < NUM_THREADS; i++) {
        threadData[i].hFile = hFile;
        threadData[i].success = true;

        HANDLE hThread = CreateThread(NULL, 0, ReadThreadProc,
&threadData[i], 0, NULL);
        threads.push_back(hThread);
    }

    WaitForMultipleObjects(NUM_THREADS, threads.data(), TRUE, INFINITE);

    for (auto h : threads) CloseHandle(h);

    for (const auto& td : threadData) {
        if (!td.success) {
            result.success = false;
            result.error_message = "Thread read error";
            CloseHandle(hFile);
            return result;
        }
    }

    CloseHandle(hFile);
    end = high_resolution_clock::now();
    result.read_time_ms = duration<double, milli>(end - start).count();

    DeleteFileA(filename);
    return result;
}

// 4. Асинхронный ввод-вывод
BenchmarkResult TestAsyncIO() {
    BenchmarkResult result;
    result.method_name = "4. Asynchronous I/O (Overlapped)";
    result.success = true;

    const char* filename = "test_async.dat";
    const size_t NUM_ASYNC = ASYNC_OPERATIONS;

    vector<vector<BYTE>> buffers(NUM_ASYNC);
    vector<OVERLAPPED> overlappeds(NUM_ASYNC);
    vector<HANDLE> events(NUM_ASYNC);

    for (size_t i = 0; i < NUM_ASYNC; i++) {
        buffers[i].resize(BUFFER_SIZE);
        for (size_t j = 0; j < BUFFER_SIZE; j++) {
            buffers[i][j] = static_cast<BYTE>((i + j) % 256);
        }

        events[i] = CreateEvent(NULL, FALSE, FALSE, NULL);
        ZeroMemory(&overlappeds[i], sizeof(OVERLAPPED));
```

```
        overlappeds[i].hEvent = events[i];
    }

    // ЗАПИСЬ
    auto start = high_resolution_clock::now();

    HANDLE hFile = CreateFileA(
        filename,
        GENERIC_WRITE,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_FLAG_OVERLAPPED,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        result.success = false;
        result.error_message = "Cannot create file";
        for (auto e : events) CloseHandle(e);
        return result;
    }

    size_t totalWritten = 0;
    size_t nextBuffer = 0;

    // Запускаем первую волну операций
    for (size_t i = 0; i < NUM_ASYNC && totalWritten < FILE_SIZE; i++) {
        overlappeds[i].Offset = static_cast<DWORD>(totalWritten &
0xFFFFFFFF);
        overlappeds[i].OffsetHigh = static_cast<DWORD>(totalWritten >>
32);

        DWORD bytesWritten;
        size_t toWrite = min(BUFFER_SIZE, FILE_SIZE - totalWritten);

        WriteFile(hFile, buffers[i].data(), static_cast<DWORD>(toWrite),
&bytesWritten, &overlappeds[i]);
        totalWritten += toWrite;
        nextBuffer++;
    }

    // Ждем завершения операций и запускаем новые
    while (nextBuffer < NUM_ASYNC || totalWritten < FILE_SIZE) {
        DWORD waitResult =
WaitForMultipleObjects(static_cast<DWORD>(min(nextBuffer, NUM_ASYNC)),
                                                    events.data(),
FALSE, INFINITE);

        size_t completedIdx = waitResult - WAIT_OBJECT_0;

        DWORD bytesWritten;
        GetOverlappedResult(hFile, &overlappeds[completedIdx],
&bytesWritten, FALSE);

        // Запускаем новую операцию, если есть что записывать
        if (totalWritten < FILE_SIZE) {
            overlappeds[completedIdx].Offset =
static_cast<DWORD>(totalWritten & 0xFFFFFFFF);
            overlappeds[completedIdx].OffsetHigh =
static_cast<DWORD>(totalWritten >> 32);

            size_t toWrite = min(BUFFER_SIZE, FILE_SIZE - totalWritten);
            WriteFile(hFile, buffers[completedIdx].data(),
static_cast<DWORD>(toWrite),
                      &bytesWritten, &overlappeds[completedIdx]);
            totalWritten += toWrite;
        } else {
            nextBuffer--;
        }
    }
```

```
    CloseHandle(hFile);
    auto end = high_resolution_clock::now();
    result.write_time_ms = duration<double, milli>(end - start).count();

    // ЧТЕНИЕ
    start = high_resolution_clock::now();

    hFile = CreateFileA(
        filename,
        GENERIC_READ,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_FLAG_OVERLAPPED,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        result.success = false;
        result.error_message = "Cannot open file";
        for (auto e : events) CloseHandle(e);
        return result;
    }

    size_t totalRead = 0;
    nextBuffer = 0;

    for (size_t i = 0; i < NUM_ASYNC && totalRead < FILE_SIZE; i++) {
        overlappeds[i].Offset = static_cast<DWORD>(totalRead &
0xFFFFFFFF);
        overlappeds[i].OffsetHigh = static_cast<DWORD>(totalRead >> 32);

        DWORD bytesRead;
        ReadFile(hFile, buffers[i].data(), BUFFER_SIZE, &bytesRead,
&overlappeds[i]);
        totalRead += min(BUFFER_SIZE, FILE_SIZE - totalRead);
        nextBuffer++;
    }

    while (nextBuffer > 0 && totalRead < FILE_SIZE) {
        DWORD waitResult =
WaitForMultipleObjects(static_cast<DWORD>(min(nextBuffer, NUM_ASYNC)),
                                                    events.data(),
FALSE, INFINITE);

        size_t completedIdx = waitResult - WAIT_OBJECT_0;

        DWORD bytesRead;
        GetOverlappedResult(hFile, &overlappeds[completedIdx],
&bytesRead, FALSE);

        if (totalRead < FILE_SIZE) {
            overlappeds[completedIdx].Offset =
static_cast<DWORD>(totalRead & 0xFFFFFFFF);
            overlappeds[completedIdx].OffsetHigh =
static_cast<DWORD>(totalRead >> 32);

            ReadFile(hFile, buffers[completedIdx].data(), BUFFER_SIZE,
&bytesRead,
                     &overlappeds[completedIdx]);
            totalRead += min(BUFFER_SIZE, FILE_SIZE - totalRead);
        } else {
            nextBuffer--;
        }
    }

    CloseHandle(hFile);
    end = high_resolution_clock::now();
    result.read_time_ms = duration<double, milli>(end - start).count();
```

```
    for (auto e : events) CloseHandle(e);
    DeleteFileA(filename);
    return result;
}

// 5. Отображение файлов в память
BenchmarkResult TestMemoryMappedIO() {
    BenchmarkResult result;
    result.method_name = "5. Memory-Mapped I/O";
    result.success = true;

    const char* filename = "test_mmap.dat";

    // ЗАПИСЬ
    auto start = high_resolution_clock::now();

    HANDLE hFile = CreateFileA(
        filename,
        GENERIC_READ | GENERIC_WRITE,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        result.success = false;
        result.error_message = "Cannot create file";
        return result;
    }

    HANDLE hMapping = CreateFileMappingA(
        hFile,
        NULL,
        PAGE_READWRITE,
        static_cast<DWORD>(FILE_SIZE >> 32),
        static_cast<DWORD>(FILE_SIZE & 0xFFFFFFFF),
        NULL
    );

    if (!hMapping) {
        result.success = false;
        result.error_message = "Cannot create file mapping";
        CloseHandle(hFile);
        return result;
    }

    BYTE* pView = (BYTE*)MapViewOfFile(
        hMapping,
        FILE_MAP_WRITE,
        0,
        0,
        FILE_SIZE
    );

    if (!pView) {
        result.success = false;
        result.error_message = "Cannot map view of file";
        CloseHandle(hMapping);
        CloseHandle(hFile);
        return result;
    }

    // Записываем данные напрямую в отображенную память
    for (size_t i = 0; i < FILE_SIZE; i++) {
        pView[i] = static_cast<BYTE>(i % 256);
    }

    // Принудительный сброс на диск
    FlushViewOfFile(pView, FILE_SIZE);
```

```
UnmapViewOfFile(pView);
CloseHandle(hMapping);
CloseHandle(hFile);

auto end = high_resolution_clock::now();
result.write_time_ms = duration<double, milli>(end - start).count();

// ЧТЕНИЕ
start = high_resolution_clock::now();

hFile = CreateFileA(
    filename,
    GENERIC_READ,
    0,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL
);

if (hFile == INVALID_HANDLE_VALUE) {
    result.success = false;
    result.error_message = "Cannot open file for reading";
    return result;
}

hMapping = CreateFileMappingA(
    hFile,
    NULL,
    PAGE_READONLY,
    static_cast<DWORD>(FILE_SIZE >> 32),
    static_cast<DWORD>(FILE_SIZE & 0xFFFFFFFF),
    NULL
);

if (!hMapping) {
    result.success = false;
    result.error_message = "Cannot create file mapping for reading";
    CloseHandle(hFile);
    return result;
}

const BYTE* pReadView = (const BYTE*)MapViewOfFile(
    hMapping,
    FILE_MAP_READ,
    0,
    0,
    FILE_SIZE
);

if (!pReadView) {
    result.success = false;
    result.error_message = "Cannot map view of file for reading";
    CloseHandle(hMapping);
    CloseHandle(hFile);
    return result;
}

// Читаем данные из отображенной памяти
volatile BYTE sum = 0;
for (size_t i = 0; i < FILE_SIZE; i++) {
    sum += pReadView[i];
}

UnmapViewOfFile(pReadView);
CloseHandle(hMapping);
CloseHandle(hFile);

end = high_resolution_clock::now();
result.read_time_ms = duration<double, milli>(end - start).count();
```

```
    DeleteFileA(filename);
    return result;
}

int main() {
    SetConsoleOutputCP(CP_UTF8);
    cout << "\nРазмер файла: " << (FILE_SIZE / (1024 * 1024)) << " MB"
<< endl;
    cout << "Размер буффера: " << (BUFFER_SIZE / 1024) << " KB" << endl;
    cout << "Колличество ядер: " << NUM_THREADS << endl;
    cout << "Асинхронные операции: " << ASYNC_OPERATIONS << endl;

    vector<BenchmarkResult> results;

    results.push_back(TestBlockingIO());
    results.push_back(TestInterruptDrivenIO());
    results.push_back(TestMultithreadedIO());
    results.push_back(TestAsyncIO());
    results.push_back(TestMemoryMappedIO());

    cout << endl;
    cout << "                                        Сравнительная таблица"
<< endl;
    cout << endl;
    cout << left << setw(35) << "Метод"
         << right << setw(15) << "  Запись (ms)    "
         << setw(15) << "   Чтение (ms)     "
         << setw(15) << "   Суммарно (ms)   " << endl;
    cout << string(80, '-') << endl;

    for (const auto& result : results) {
        if (result.success) {
            cout << left << setw(35) << result.method_name
                 << right << setw(15) << fixed << setprecision(2) <<
result.write_time_ms
                 << setw(15) << result.read_time_ms
                 << setw(15) << (result.write_time_ms +
result.read_time_ms) << endl;
        }
    }

    cin.get();

    return 0;
}
```