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 №2
на тему:
«РАБОТА С ФАЙЛАМИ»
БГУИР 6-05-0612-02 67

Выполнил студент группы 353503
КОХАН Артём Игоревич

(дата, подпись студента)

Проверил ассистент каф. информатики
Гриценко Никита Юрьевич

(дата, подпись преподавателя)

Минск 2025

СОДЕРЖАНИЕ

1 ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ

Целью выполнения лабораторной работы является сравнительный анализ эффективности двух методов обработки файлов в операционной системе Windows с использованием многопоточности. В ходе выполнения работы необходимо изучить и практически сравнить традиционный метод работы с файлами (посредством функций ReadFile/WriteFile) и метод отображения файлов в память (File Mapping).

Задача лабораторной работы заключается в разработке многопоточного приложения, которое выполняет обработку данных файла с использованием обоих методов при различном количестве потоков выполнения. Программа должна выполнять следующие функции:

1 Создание тестового файла: генерация файла заданного размера со тестовыми данными для обеспечения одинаковых условий тестирования.

2 Реализация традиционного метода обработки: чтение файла в буфер оперативной памяти, многопоточная обработка данных, запись изменённых данных обратно в файл.

3 Реализация метода отображения в память: создание файлового отображения, работа с данными непосредственно через указатель на отображенную область, многопоточная обработка.

4 Сравнительный анализ производительности: измерение времени выполнения операций для каждого метода при разном количестве потоков (1, 2, 4, 8).

5 Визуализация результатов: вывод сравнительной таблицы с временем выполнения и расчет коэффициента ускорения для наглядной демонстрации эффективности методов.

Приложение должно продемонстрировать преимущества и недостатки каждого подхода, выявить оптимальное количество потоков для различных методов обработки и показать влияние многопоточности на производительность операций ввода-вывода.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Традиционный метод обработки файлов в Windows основан на использовании функций ReadFile и WriteFile. Данный подход предполагает последовательное чтение данных из файла в буфер оперативной памяти, обработку данных и последующую запись изменений обратно в файл [1].

Основные этапы традиционного метода:

1 Открытие файла с помощью CreateFile.

2 Чтение данных в буфер через ReadFile.

3 Обработка данных в памяти.

4 Запись обработанных данных через WriteFile.

5 Закрытие файла.

Преимуществом данного метода является простота реализации, однако при работе с большими файлами возникают накладные расходы на копирование данных между диском и памятью.

Метод отображения файлов в память позволяет работать с файлом как с массивом байтов в виртуальной памяти процесса. Операционная система автоматически подгружает необходимые участки файла в память и синхронизирует изменения с диском [2].

Основные этапы работы с File Mapping:

1 Создание файлового объекта (CreateFile).

2 Создание объекта отображения (CreateFileMapping).

3 Отображение файла в память (MapViewOfFile).

4 Непосредственная работа с данными через указатель.

5 Освобождение ресурсов (UnmapViewOfFile, CloseHandle).

Основное преимущество – исключение промежуточного копирования данных и возможность эффективной работы с большими файлами.

Многопоточность позволяет распределить обработку файла между несколькими потоками выполнения. Каждый поток обрабатывает свою часть данных, что может значительно ускорить выполнение операции на многопроцессорных системах [3].

Ключевые аспекты многопоточной обработки:

1 Разделение файла на блоки для параллельной обработки.

2 Синхронизация доступа к общим ресурсам.

3 Оптимальное количество потоков (обычно равно количеству процессорных ядер).

4 Минимизация конфликтов при доступе к данным.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Согласно формулировке задачи, были спроектированы следующие функции программы:

– ProcessData;
– ProcessFileTraditionalMultiThreaded;
– ProcessFileWithMappingMultiThreaded;
– TestWithDifferentThreadCounts;
– main.

3.1 Функция ProcessData

Данная функция выполняет обработку данных в многопоточном режиме. Функция совершает следующие действия:

1 Принимает указатель на данные, начальный и конечный индекс обрабатываемого блока.

2 Выполняет побайтовую обработку данных (инвертирование битов).

3 Обеспечивает изолированную работу каждого потока с своей portion данных.

3.2 Функция ProcessFileTraditionalMultiThreaded

Функция реализует традиционный метод обработки файла с использованием многопоточности. Функция совершает следующие действия:

1 Открывает файл для чтения и определяет его размер.

2 Выделяет буфер в оперативной памяти и читает весь файл.

3 Создает заданное количество потоков для параллельной обработки данных.

4 Каждому потоку назначает свой блок данных для обработки.

5 Записывает обработанные данные обратно в файл.

6 Освобождает все allocated ресурсы.

3.3 Функция ProcessFileWithMappingMultiThreaded

Функция реализует метод отображения файла в память с многопоточной обработкой. Функция совершает следующие действия:

1 Создает файловое отображение с помощью CreateFileMapping.

2 Отображает файл в виртуальную память процесса через MapViewOfFile.

3 Организует многопоточную обработку данных непосредственно через указатель на отображенную область.

4 Обеспечивает автоматическую синхронизацию изменений с файлом на диске.

5 Корректно освобождает ресурсы файлового отображения.

3.4 Функция TestWithDifferentThreadCounts

Функция выполняет сравнительное тестирование обоих методов. Функция совершает следующие действия:

1 Организует тестирование для разного количества потоков (1, 2, 4, 8).

2 Для каждого теста создает копии исходного файла.

3 Измеряет время выполнения для традиционного метода и метода отображения.

4 Формирует сравнительную таблицу результатов.

5 Вычисляет коэффициент ускорения и определяет оптимальное количество потоков.

3.5 Функция main

Главная функция программы совершает следующие действия:

1 Инициализирует консоль для работы с русским языком.

2 Создает тестовый файл заданного размера (100 МБ).

3 Запускает сравнительное тестирование методов обработки.

4 Выводит аналитические выводы по результатам тестирования.

5 Обеспечивает корректное завершение работы программы.

4 ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.1 Запуск программы и процесс выполнения

Программа начинает работу с создания тестового файла размером 100 МБ, заполненного тестовыми данными. После этого последовательно выполняются тесты для разного количества потоков (1, 2, 4, 8).

Для каждого количества потоков программа:

– Создает копии исходного файла для тестирования обоих методов;

– Замеряет время выполнения традиционного метода;

– Замеряет время выполнения метода отображения в память;

– Выводит промежуточные результаты в консоль.

```
=== РЕЗУЛЬТАТЫ СРАВНЕНИЯ ===
============================================================
| Количество   | Традиционный    | Отображение     | Выигрыш  |
| потоков      | метод (сек)     | в память (сек)  |          |
============================================================
|            1 |           0.599 |           0.484 |     1.24x |
|            2 |           0.445 |           0.268 |     1.66x |
|            4 |           0.370 |           0.180 |     2.06x |
|            8 |           0.358 |           0.137 |     2.61x |
============================================================

=== АНАЛИЗ РЕЗУЛЬТАТОВ ===
Лучшая производительность традиционного метода: 8 потоков (0.358 сек)
Лучшая производительность отображения в память: 8 потоков (0.137 сек)
Максимальное ускорение: 2.61x
```

Рисунок 4.1 – Результат выполнения программы и анализ результатов

4.2 Описание работы и результатов

В процессе выполнения программа демонстрирует сравнительную производительность двух методов обработки файлов. Результаты тестирования показывают, что метод отображения файлов в память в большинстве случаев обеспечивает более высокую производительность по сравнению с традиционным подходом.

Ключевые наблюдения:

– Метод отображения в память показывает лучшее время выполнения при работе с большими файлами;

– Оптимальное количество потоков зависит от конкретного метода и аппаратных характеристик системы;

– При увеличении количества потоков свыше оптимального значения могут возникать накладные расходы на синхронизацию;

– Традиционный метод может быть более эффективен при работе с небольшими файлами или при определенных паттернах доступа.

ВЫВОД

В ходе выполнения лабораторной работы была успешно разработана многопоточная консольная программа на языке C++, предназначенная для сравнительного анализа методов обработки файлов в среде Windows. Программа реализует два подхода: традиционный метод с использованием функций ReadFile/WriteFile и метод отображения файлов в память (File Mapping).

Результаты тестирования показали, что метод отображения файлов в память в среднем демонстрирует более высокую производительность при обработке больших файлов. Это объясняется исключением промежуточного копирования данных и уменьшением количества системных вызовов.

Также было установлено, что оптимальное количество потоков для обработки файлов зависит от конкретного метода и составляет обычно 2-4 потока для типичных многопроцессорных систем. Дальнейшее увеличение количества потоков не всегда приводит к росту производительности из-за накладных расходов на синхронизацию и управление потоками.

Практическая значимость работы заключается в получении опыта разработки эффективных многопоточных приложений для работы с файлами и понимании преимуществ различных подходов к обработке данных в операционной системе Windows.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Понятие файла и кластера [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/fileio/files-and-clusters. – Дата доступа: 18.10.2025.

[2] File Mapping [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/memory/file-mapping. – Дата доступа: 18.10.2025.

[3] Процессы и потоки [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/procthread/processes-and-threads. – Дата доступа: 18.10.2025.

ПРИЛОЖЕНИЕ А
(справочное)
Исходный код

```
#include <windows.h>
#include <stdio.h>
#include <time.h>
#include <iostream>
#include <vector>
#include <thread>
#include <string>

// Функция для обработки данных (простой пример - инвертирование байтов)
void ProcessData(BYTE* data, DWORD start, DWORD end) {
    for (DWORD i = start; i < end; i++) {
        data[i] = ~data[i]; // Инвертируем каждый байт
    }
}

// Многопоточная обработка для традиционного метода
BOOL ProcessFileTraditionalMultiThreaded(LPCSTR filename, int num_threads) {
    HANDLE hFile = INVALID_HANDLE_VALUE;
    BYTE* buffer = NULL;
    DWORD fileSize = 0;
    DWORD bytesRead = 0;
    DWORD bytesWritten = 0;

    printf("Традиционный метод (%d потоков): %s\n", num_threads, filename);

    // 1. Открываем файл для чтения
    hFile = CreateFileA(
        filename,
        GENERIC_READ,
        FILE_SHARE_READ,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Ошибка: не удалось открыть файл для чтения\n");
        return FALSE;
    }

    // 2. Получаем размер файла
    fileSize = GetFileSize(hFile, NULL);
    if (fileSize == INVALID_FILE_SIZE) {
        printf("Ошибка: не удалось получить размер файла\n");
        CloseHandle(hFile);
        return FALSE;
    }

    printf("Размер файла: %lu байт\n", fileSize);

    if (fileSize == 0) {
        printf("Файл пуст\n");
        CloseHandle(hFile);
        return TRUE;
    }

    // 3. Выделяем память под данные
```

```
    buffer = (BYTE*)malloc(fileSize);
    if (buffer == NULL) {
        printf("Ошибка: не удалось выделить память\n");
        CloseHandle(hFile);
        return FALSE;
    }

    // 4. Читаем файл
    if (!ReadFile(hFile, buffer, fileSize, &bytesRead, NULL) || bytesRead !=
fileSize) {
        printf("Ошибка: не удалось прочитать файл\n");
        free(buffer);
        CloseHandle(hFile);
        return FALSE;
    }

    CloseHandle(hFile);

    // 5. Многопоточная обработка данных
    printf("Начало многопоточной обработки данных (%d потоков)...\n",
num_threads);

    std::vector<std::thread> threads;
    DWORD chunk_size = fileSize / num_threads;

    for (int i = 0; i < num_threads; i++) {
        DWORD start = i * chunk_size;
        DWORD end = (i == num_threads - 1) ? fileSize : (i + 1) * chunk_size;

        threads.emplace_back([buffer, start, end, i]() {
            printf("Поток %d обрабатывает байты [%lu - %lu]\n", i, start, end
- 1);
            ProcessData(buffer, start, end);
            });
    }

    // Ждем завершения всех потоков
    for (auto& thread : threads) {
        thread.join();
    }

    printf("Многопоточная обработка данных завершена\n");

    // 6. Открываем файл для записи
    hFile = CreateFileA(
        filename,
        GENERIC_WRITE,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Ошибка: не удалось открыть файл для записи\n");
        free(buffer);
        return FALSE;
    }

    // 7. Записываем обработанные данные
    if (!WriteFile(hFile, buffer, fileSize, &bytesWritten, NULL) ||
bytesWritten != fileSize) {
        printf("Ошибка: не удалось записать файл\n");
```

```
    }

    // 8. Освобождаем ресурсы
    free(buffer);
    CloseHandle(hFile);

    return TRUE;
}

// Многопоточная обработка для метода отображения в память
BOOL ProcessFileWithMappingMultiThreaded(LPCSTR filename, int num_threads) {
    HANDLE hFile = INVALID_HANDLE_VALUE;
    HANDLE hMapping = NULL;
    BYTE* pData = NULL;
    DWORD fileSize = 0;

    printf("Отображение в память (%d потоков): %s\n", num_threads, filename);

    // 1. Открываем файл
    hFile = CreateFileA(
        filename,
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Ошибка: не удалось открыть файл. Код ошибки: %lu\n",
GetLastError());
        return FALSE;
    }

    // 2. Получаем размер файла
    fileSize = GetFileSize(hFile, NULL);
    if (fileSize == INVALID_FILE_SIZE) {
        printf("Ошибка: не удалось получить размер файла\n");
        CloseHandle(hFile);
        return FALSE;
    }

    printf("Размер файла: %lu байт\n", fileSize);

    if (fileSize == 0) {
        printf("Файл пуст\n");
        CloseHandle(hFile);
        return TRUE;
    }

    // 3. Создаем файловое отображение
    hMapping = CreateFileMapping(
        hFile,
        NULL,
        PAGE_READWRITE,
        0,
        0,
        NULL
    );

    if (hMapping == NULL) {
        printf("Ошибка: не удалось создать файловое отображение. Код ошибки:
%lu\n", GetLastError());
```

```
        CloseHandle(hFile);
        return FALSE;
    }

    // 4. Отображаем файл в память
    pData = (BYTE*)MapViewOfFile(
        hMapping,
        FILE_MAP_ALL_ACCESS,
        0,
        0,
        0
    );

    if (pData == NULL) {
        printf("Ошибка: не удалось отобразить файл в память. Код ошибки: %lu\n",
GetLastError());
        CloseHandle(hMapping);
        CloseHandle(hFile);
        return FALSE;
    }

    printf("Файл успешно отображен в память по адресу: %p\n", pData);

    // 5. Многопоточная обработка данных
    printf("Начало многопоточной обработки данных (%d потоков)...\n",
num_threads);

    std::vector<std::thread> threads;
    DWORD chunk_size = fileSize / num_threads;

    for (int i = 0; i < num_threads; i++) {
        DWORD start = i * chunk_size;
        DWORD end = (i == num_threads - 1) ? fileSize : (i + 1) * chunk_size;

        threads.emplace_back([pData, start, end, i]() {
            printf("Поток %d обрабатывает байты [%lu - %lu]\n", i, start, end
- 1);
            ProcessData(pData, start, end);
            });
    }

    // Ждем завершения всех потоков
    for (auto& thread : threads) {
        thread.join();
    }

    printf("Многопоточная обработка данных завершена\n");

    // 6. Освобождаем ресурсы
    if (!UnmapViewOfFile(pData)) {
        printf("Предупреждение: ошибка при освобождении отображения\n");
    }

    CloseHandle(hMapping);
    CloseHandle(hFile);

    return TRUE;
}

// Функция для создания тестового файла
BOOL CreateTestFile(LPCSTR filename, DWORD size) {
    HANDLE hFile = CreateFileA(
        filename,
        GENERIC_WRITE,
```

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

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Ошибка создания тестового файла\n");
        return FALSE;
    }

    // Заполняем файл тестовыми данными
    BYTE* testData = (BYTE*)malloc(size);
    if (testData) {
        for (DWORD i = 0; i < size; i++) {
            testData[i] = (BYTE)(i % 256);
        }

        DWORD bytesWritten;
        WriteFile(hFile, testData, size, &bytesWritten, NULL);
        free(testData);
    }

    CloseHandle(hFile);
    printf("Создан тестовый файл: %s, размер: %lu байт\n", filename, size);
    return TRUE;
}

// Функция для копирования файла (для тестирования на одинаковых данных)
BOOL CopyTestFile(LPCSTR source, LPCSTR dest) {
    return CopyFileA(source, dest, FALSE);
}

// Тестирование методов с разным количеством потоков
void TestWithDifferentThreadCounts(const char* base_filename, DWORD fileSize)
{
    std::vector<int> thread_counts = { 1, 2, 4, 8 };

    printf("\n=== СРАВНЕНИЕ МЕТОДОВ С РАЗНЫМ КОЛИЧЕСТВОМ ПОТОКОВ ===\n");
    printf("Размер файла: %lu MB\n\n", fileSize / (1024 * 1024));

    // Таблица результатов
    struct Result {
        double traditional_time;
        double mapping_time;
    };

    std::vector<Result> results(thread_counts.size());

    // Тестируем для каждого количества потоков
    for (size_t i = 0; i < thread_counts.size(); i++) {
        int threads = thread_counts[i];
        printf("--- Тестирование с %d потоками ---\n", threads);

        // Создаем копии файлов для тестирования
        char traditional_filename[MAX_PATH];
        char mapping_filename[MAX_PATH];

        sprintf_s(traditional_filename, "traditional_%d.dat", threads);
        sprintf_s(mapping_filename, "mapping_%d.dat", threads);

        CopyTestFile(base_filename, traditional_filename);
        CopyTestFile(base_filename, mapping_filename);
```

```
        // Тестируем традиционный метод
        printf("Традиционный метод...\n");
        clock_t start = clock();
        ProcessFileTraditionalMultiThreaded(traditional_filename, threads);
        clock_t end = clock();
        results[i].traditional_time = ((double)(end - start)) / CLOCKS_PER_SEC;

        // Тестируем метод отображения в память
        printf("Отображение в память...\n");
        start = clock();
        ProcessFileWithMappingMultiThreaded(mapping_filename, threads);
        end = clock();
        results[i].mapping_time = ((double)(end - start)) / CLOCKS_PER_SEC;

        // Удаляем временные файлы
        DeleteFileA(traditional_filename);
        DeleteFileA(mapping_filename);

        printf("Завершено\n\n");
    }

    // Выводим сравнительную таблицу (исправленную)
    printf("=== РЕЗУЛЬТАТЫ СРАВНЕНИЯ ===\n");

printf("================================================================\n");
    printf("| Количество   | Традиционный    | Отображение     | Выигрыш  |\n");
    printf("| потоков      | метод (сек)     | в память (сек)  |          |\n");

printf("================================================================\n");

    for (size_t i = 0; i < thread_counts.size(); i++) {
        double traditional = results[i].traditional_time;
        double mapping = results[i].mapping_time;
        double advantage = traditional / mapping;

        printf("| %12d | %15.3f | %15.3f | %8.2fx |\n",
            thread_counts[i], traditional, mapping, advantage);
    }

printf("================================================================\n");

    // Анализ результатов
    printf("\n=== АНАЛИЗ РЕЗУЛЬТАТОВ ===\n");
    double best_traditional = results[0].traditional_time;
    double best_mapping = results[0].mapping_time;
    int best_traditional_threads = 1;
    int best_mapping_threads = 1;

    for (size_t i = 1; i < thread_counts.size(); i++) {
        if (results[i].traditional_time < best_traditional) {
            best_traditional = results[i].traditional_time;
            best_traditional_threads = thread_counts[i];
        }
        if (results[i].mapping_time < best_mapping) {
            best_mapping = results[i].mapping_time;
            best_mapping_threads = thread_counts[i];
        }
    }

    printf("Лучшая производительность традиционного метода: %d потоков (%.3f
сек)\n",
        best_traditional_threads, best_traditional);
```

```
    printf("Лучшая производительность отображения в память: %d потоков (%.3f
сек)\n",
        best_mapping_threads, best_mapping);
    printf("Максимальное ускорение: %.2fx\n", best_traditional / best_mapping);
}

int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    const char* base_filename = "base_test_file.bin";
    const DWORD fileSize = 100 * 1024 * 1024;

    printf("=== СРАВНИТЕЛЬНЫЙ АНАЛИЗ МЕТОДОВ ОБРАБОТКИ ФАЙЛОВ ===\n");

    // Создаем базовый тестовый файл
    if (!CreateTestFile(base_filename, fileSize)) {
        printf("Не удалось создать тестовый файл\n");
        return 1;
    }

    // Тестируем оба метода с разным количеством потоков
    TestWithDifferentThreadCounts(base_filename, fileSize);

    DeleteFileA(base_filename);

    printf("\nНажмите Enter для выхода...");
    getchar();

    return 0;
}
```