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**

Выполнил: студент гр.253504 Новиков В.А.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc178641666)

[2 Краткие теоритические сведения 4](#_Toc178641667)

[3 Описание функций программы 5](#_Toc178641668)

[Заключение 7](#_Toc178641669)

[Список использованных источников 8](#_Toc178641670)

[Приложение А (обязательное) исходный код продукта 9](#_Toc178641671)

# **1 ФОРМУЛИРОВКА ЗАДАЧИ**

Данная лабораторная работа посвящена изучению расширенных методов работы с файлами в системном программировании, таких как асинхронный ввод-вывод и отображение файлов в память. Цель работы заключается в демонстрации возможностей асинхронных операций с файлами и оценке их эффективности по сравнению с традиционным (синхронным) подходом к вводу-выводу.

В рамках задачи необходимо разработать приложение, которое выполняет обработку содержимого файла, используя асинхронный ввод-вывод. Асинхронный метод предполагает параллельное выполнение операций чтения, обработки и записи данных, что позволяет увеличить производительность за счет выполнения нескольких операций одновременно, не блокируя основной поток выполнения программы.

Для реализации асинхронного ввода-вывода будут использоваться механизмы, такие как структура *OVERLAPPED* и функции *ReadFileEx()* и *WriteFileEx()*, которые позволяют запускать операции ввода-вывода без ожидания их завершения. Необходимо также варьировать количество параллельно инициированных операций ввода-вывода, чтобы оценить влияние этого параметра на общую производительность программы и найти потенциальные «узкие места» в реализации, которые могут ограничивать эффективность.

Задача также включает оценку производительности разработанного решения по сравнению с традиционным синхронным методом, при котором операции чтения, обработки и записи выполняются последовательно. Для получения точных и измеримых результатов следует использовать достаточно большой объем данных, чтобы длительность выполнения операций была заметной.

Дополнительно, возможно сравнение производительности асинхронной реализации с многопоточной, где каждая операция (чтение, обработка, запись) может выполняться в отдельном потоке. Варианты обработки данных могут включать сортировку числовых данных, статистический анализ, криптографическую обработку и другие методы, которые требуют длительного времени для выполнения.

Таким образом, данная работа предоставляет возможность изучить и сравнить различные подходы к организации ввода-вывода в системном программировании, оценить их эффективность и выявить наиболее оптимальные решения для обработки больших объемов данных.

**2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ**

Файловая система — это структура, организующая хранение и доступ к данным на различных информационных носителях, таких как жесткие диски, флеш-накопители или сетевые устройства хранения. Она определяет правила именования файлов, их расположение на носителе и способы взаимодействия программного обеспечения с данными. Файловая система также предоставляет *API* для приложений, с помощью которых можно открывать, читать, записывать и закрывать файлы. [1]

Операции ввода-вывода — это один из ключевых аспектов взаимодействия программ с файлами и внешними устройствами. В системах ввода-вывода можно выделить два основных типа взаимодействия: синхронный и асинхронный ввод-вывод. При синхронном вводе-выводе приложение, вызвав операцию чтения или записи, ожидает завершения этой операции перед продолжением выполнения дальнейших инструкций. Это может привести к блокировке потока, особенно если операция ввода-вывода выполняется медленно (например, чтение большого файла или взаимодействие с сетью). Синхронный подход удобен в простых сценариях, где требуется последовательное выполнение действий, но может существенно снизить производительность, если операции ввода-вывода занимают значительное время.

Асинхронный ввод-вывод, также называемый перекрывающимся, позволяет программе продолжать выполнение других задач, пока операция чтения или записи выполняется в фоновом режиме. Этот механизм работает благодаря структуре *OVERLAPPED*, которая используется для передачи данных и отслеживания состояния операции. Асинхронные функции, такие как *ReadFileEx()* и *WriteFileEx()*, позволяют инициализировать ввод-вывод без ожидания завершения, освобождая поток для выполнения других задач. По завершении операции операционная система уведомляет приложение о том, что данные были обработаны.

Отображение файлов в память — это еще один способ работы с файлами, который позволяет проецировать содержимое файла на область памяти, делая доступ к данным быстрым и удобным. Вместо последовательного чтения и записи данных можно работать с ними напрямую как с массивом в оперативной памяти. Функции *CreateFileMapping()* и *MapViewOfFile()* используются для создания и отображения файлов в память. [2]

Таким образом, выбор подхода к вводу-выводу зависит от конкретной задачи. Асинхронный ввод-вывод позволяет улучшить производительность в сценариях, где важно параллельно выполнять обработку данных и ввод-вывод. Синхронные операции могут быть более эффективными для простых и небольших задач. Мультиплексирование ввода-вывода и отображение файлов в память предоставляют дополнительные возможности для оптимизации работы с большими объемами данных.

# **3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ**

В данной программе реализуются функции для обработки содержимого файла с использованием асинхронного ввода-вывода, включая сортировку чисел и запись отсортированных данных в выходной файл. Программа состоит из нескольких ключевых функций.

*«void GenerateTestFile(const char\* filename, size\_t dataSize)»*:Создает тестовый файл с указанным количеством случайных данных. Этот файл используется для проверки работы программы. Файл открывается в режиме записи с помощью *CreateFile*, генерируется вектор случайных чисел размером *dataSize*. Эти числа записываются в файл с помощью *WriteFile*, после чего файл закрывается.

*«void AsyncFileProcessing(const char\* filename)»*: Выполняет асинхронное чтение, обработку (сортировку) и запись данных обратно в файл с использованием *ReadFileEx* и *WriteFileEx*. Файл открывается в асинхронном режиме с флагами *GENERIC\_READ*, *GENERIC\_WRITE* и *FILE\_FLAG\_OVERLAPPED*. Создается массив структур *AsyncContext*, каждая из которых содержит буфер для данных, структуру *OVERLAPPED*, указывающую смещение для чтения/записи и дескриптор файла. Цикл инициализирует несколько асинхронных операций чтения с помощью *ReadFileEx*, передавая каждую *OVERLAPPED* структуру и буфер. В каждом асинхронном чтении при завершении вызывается *callback*-функция *ReadCompletionRoutine*. В *ReadCompletionRoutine* данные сортируются, и сразу начинается асинхронная запись с использованием *WriteFileEx*. Основной поток программы использует *SleepEx* в цикле для ожидания завершения всех асинхронных операций.

*«void CALLBACK ReadCompletionRoutine(DWORD errorCode, DWORD bytesTransferred, LPOVERLAPPED lpOverlapped)»*: *Callback*-функция, которая вызывается после завершения асинхронного чтения. Проверяет наличие ошибок. Если ошибок нет, получает *AsyncContext*, связанный с *lpOverlapped*. Сортирует данные, содержащиеся в буфере *AsyncContext*. Запускает асинхронную запись этих данных обратно в файл с помощью *WriteFileEx*, используя ту же *OVERLAPPED* структуру.

*«void SyncFileProcessing(const char\* filename)»*: Выполняет синхронное чтение, обработку (сортировку) и запись данных обратно в файл. Файл открывается в синхронном режиме, выполняется чтение данных блоками фиксированного размера в буфер. Данные в буфере сортируются, устанавливается указатель файла на начальную позицию чтения с помощью *SetFilePointerEx*, после чего данные записываются обратно в файл. Процесс продолжается, пока не будут обработаны все данные.

*«int main()»*: Основная функция программы, которая запускает асинхронные и синхронные версии обработки файла, измеряет их время выполнения и выводит результаты. Создает тестовый файл с данными с помощью *GenerateTestFile*, Запускает асинхронную обработку файла *AsyncFileProcessing* и измеряет время выполнения, запускает синхронную обработку файла *SyncFileProcessing* и также измеряет время выполнения, выводит время, затраченное на асинхронную и синхронную обработку. Результат выполнения программы на рисунке 3.1.

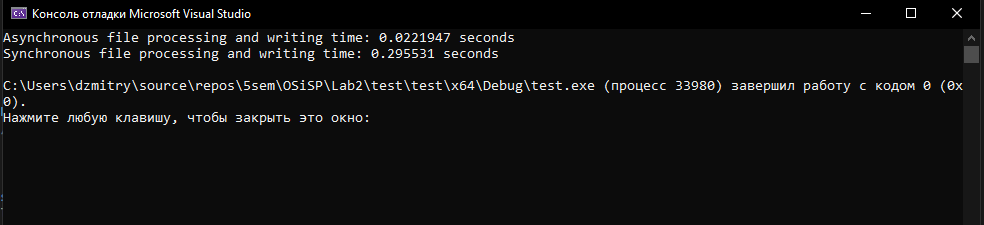


Рисунок 3.1 – Результат выполнения программы.

Результаты асинхронного выполнения показывают, что время, затраченное на асинхронные операции ввода-вывода занимают меньше времени, чем операции синхронного ввода-вывода. Однако результат во многом зависит от размера буфера. В результатах приведенных выше в операции асинхронной обработки файла использовался буфер размером 1 048 576 байт. При использовании буфера меньшего размера, например, 1024 байт, синхронные операции будут отрабатывать быстрее. Результат при меньшем буфере показан на рисунке 3.2.

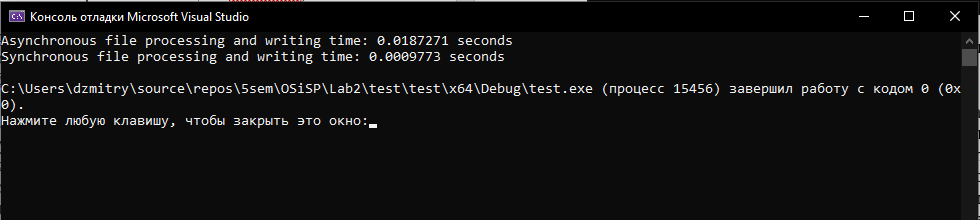


Рисунок 3.2 – Результат выполнения программы с меньшим буфером.

Синхронный ввод-вывод не позволяет эффективно выполнять обработку больших объемов данных. Он блокирует поток, ожидая завершения каждой операции ввода-вывода, что может привести к значительным задержкам в более сложных сценариях.

# **ЗАКЛЮЧЕНИЕ**

В ходе исследования, проведенного с использованием асинхронного и синхронного ввода-вывода, были получены ценные результаты, которые подчеркивают важность выбора правильного метода для обработки больших объемов данных. Асинхронный ввод-вывод демонстрирует явные преимущества в контексте производительности, позволяя значительно сократить общее время выполнения операций по сравнению с синхронными методами.

Асинхронные операции имеют множество достоинств, включая возможность выполнения других задач в параллельном режиме. Это делает их особенно привлекательными для приложений, требующих высокой производительности и быстрого отклика. В условиях значительных объемов данных асинхронный подход позволяет не только эффективно управлять временем выполнения, но и оптимизировать использование ресурсов системы.

Результаты выполнения показали, что выбор подходящего размера буфера играет критическую роль в оптимизации производительности асинхронных операций. При использовании небольших буферов наблюдается увеличение временных затрат из-за частых обращений к файловой системе, что негативно сказывается на общей производительности. С другой стороны, использование более крупных буферов значительно снижает количество операций ввода-вывода, что в итоге приводит к ускорению обработки данных.

В то время как синхронный ввод-вывод может быть эффективен для простых и малых по объему задач, он имеет свои ограничения, особенно в условиях высокой нагрузки. Его блокирующая природа делает его менее подходящим для приложений, требующих высокой степени параллелизма и быстрого отклика.

Таким образом, асинхронный подход, в сочетании с оптимизированным размером буфера, демонстрирует свою экономическую эффективность и производительность, становясь лучшим выбором для задач, требующих обработки больших объемов данных. Результаты исследования подтверждают, что внедрение асинхронных методов может существенно повысить эффективность работы приложений, что является критически важным аспектом в современном программировании и системной архитектуре.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Файловые системы Windows: виды и особенности [Электронный ресурс]. – Режим доступа: https://otus.ru/journal/

[2] Синхронный и асинхронный ввод-вывод [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/windows/win32/fileio/

# **ПРИЛОЖЕНИЕ А** **(обязательное)** **Исходный код программного продукта**

#include <windows.h>

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

const int BUFFER\_SIZE = 1024 \* 1024;

const int NUM\_ASYNC\_OPERATIONS = 4;

struct AsyncContext {

OVERLAPPED overlapped;

std::vector<int> buffer;

HANDLE hFile;

};

void GenerateTestFile(const wchar\_t\* filename, size\_t dataSize) {

HANDLE hFile = CreateFile(filename, GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);

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

std::cerr << "Error creating test file." << std::endl;

return;

}

std::vector<int> data(dataSize / sizeof(int));

for (auto& val : data) val = rand();

DWORD written;

WriteFile(hFile, data.data(), dataSize, &written, NULL);

CloseHandle(hFile);

}

void CALLBACK ReadCompletionRoutine(DWORD errorCode, DWORD bytesTransferred, LPOVERLAPPED lpOverlapped) {

if (errorCode != 0) {

std::cerr << "Read failed with error code: " << errorCode << std::endl;

return;

}

AsyncContext\* context = reinterpret\_cast<AsyncContext\*>(lpOverlapped);

size\_t elementsRead = bytesTransferred / sizeof(int);

std::sort(context->buffer.begin(), context->buffer.begin() + elementsRead);

WriteFileEx(context->hFile, context->buffer.data(), bytesTransferred, &context->overlapped, [](DWORD errorCode, DWORD bytesWritten, LPOVERLAPPED lpOverlapped) {

if (errorCode != 0) {

std::cerr << "Write failed with error code: " << errorCode << std::endl;

}

});

}

void AsyncFileProcessing(const wchar\_t\* filename) {

HANDLE hFile = CreateFile(filename, GENERIC\_READ | GENERIC\_WRITE, 0, NULL, OPEN\_EXISTING, FILE\_FLAG\_OVERLAPPED | FILE\_FLAG\_NO\_BUFFERING, NULL);

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

std::cerr << "Error opening file for async reading and writing." << std::endl;

return;

}

std::vector<AsyncContext> contexts(NUM\_ASYNC\_OPERATIONS);

size\_t offset = 0;

for (int i = 0; i < NUM\_ASYNC\_OPERATIONS; ++i) {

contexts[i].buffer.resize(BUFFER\_SIZE / sizeof(int));

contexts[i].hFile = hFile;

contexts[i].overlapped = { 0 };

contexts[i].overlapped.Offset = offset & 0xFFFFFFFF;

contexts[i].overlapped.OffsetHigh = (offset >> 32) & 0xFFFFFFFF;

if (!ReadFileEx(hFile, contexts[i].buffer.data(), BUFFER\_SIZE, &contexts[i].overlapped, ReadCompletionRoutine)) {

std::cerr << "Error starting async read operation." << std::endl;

CloseHandle(hFile);

return;

}

offset += BUFFER\_SIZE;

}

while (HasOverlappedIoCompleted(&contexts.back().overlapped) == 0) {

SleepEx(INFINITE, TRUE);

}

CloseHandle(hFile);

}

void SyncFileProcessing(const wchar\_t\* filename) {

HANDLE hFile = CreateFile(filename, GENERIC\_READ | GENERIC\_WRITE, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

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

std::cerr << "Error opening file for sync reading and writing." << std::endl;

return;

}

std::vector<int> buffer(BUFFER\_SIZE / sizeof(int));

DWORD bytesRead, bytesWritten;

LARGE\_INTEGER offset = { 0 };

while (ReadFile(hFile, buffer.data(), BUFFER\_SIZE, &bytesRead, NULL) && bytesRead > 0) {

std::sort(buffer.begin(), buffer.begin() + bytesRead / sizeof(int));

SetFilePointerEx(hFile, offset, NULL, FILE\_BEGIN);

if (!WriteFile(hFile, buffer.data(), bytesRead, &bytesWritten, NULL) || bytesWritten != bytesRead) {

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

CloseHandle(hFile);

return;

}

offset.QuadPart += bytesRead;

}

CloseHandle(hFile);

}

int main() {

const wchar\_t\* filename = L"test\_data.bin";

size\_t dataSize = 1 \* BUFFER\_SIZE;

GenerateTestFile(filename, dataSize);

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

AsyncFileProcessing(filename);

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

std::chrono::duration<double> durationAsync = endAsync - startAsync;

std::cout << "Asynchronous file processing and writing time: " << durationAsync.count() << " seconds" << std::endl;

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

SyncFileProcessing(filename);

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

std::chrono::duration<double> durationSync = endSync - startSync;

std::cout << "Synchronous file processing and writing time: " << durationSync.count() << " seconds" << std::endl;

return 0;

}