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.253503 Царук В.А.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc181121833)

[2 Краткие теоритические сведения 4](#_Toc181121834)

[3 Описание функций программы 5](#_Toc181121835)

[Заключение 7](#_Toc181121836)

[Список использованных источников 8](#_Toc181121837)

[Приложение А (обязательное) Исходный код программного продукта 9](#_Toc181121838)

# **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.

*«insertionSort(std::vector<int>& array, DWORD& iterationCount*)*»*: эта функция реализует алгоритм сортировки вставками для отсортировки вектора целых чисел. Она принимает ссылку на вектор *array*, и ссылку на переменную *iterationCount*, которая отслеживает количество выполненных итераций. Алгоритм проходит по элементам массива, сравнивает их и перемещает их в правильную позицию, увеличивая счетчик итераций на каждой перестановке.

*«*generateFile(const std::wstring& filename, int count)*»*: функция отвечает за создание входного файла с заданным именем и заполнение его числами в порядке убывания. Она принимает имя файла *filename* и количество чисел count, которые нужно записать в файл. После открытия файла в режиме записи, функция генерирует числа и записывает их в файл.

*«writeToFile(const std::vector<int>& numbers, const std::wstring& filename)»*: эта функция записывает отсортированный вектор чисел numbers в выходной файл с заданным именем *filename*. Для записи используется *API* *Windows* *CreateFile*, а также *WriteFile*, что позволяет выполнять операции записи в файл. Функция обрабатывает возможные ошибки при открытии файла и записи данных.

*«CallbackWhenReadCompletion(DWORD dwErrorCode, DWORD numberOfBytesTransferred, LPOVERLAPPED lpOverlapped)»*: это обратный вызов, который вызывается после завершения асинхронной операции чтения из файла. Функция обрабатывает возможные ошибки, считывает данные из буфера и вызывает функцию *insertionSort* для сортировки полученных чисел. Затем отсортированные данные записываются в выходной файл с помощью функции *writeToFile*. Здесь также измеряется время, затраченное на сортировку и запись.

*«fileOperationsAsync(int bufferSize)»*: эта функция выполняет асинхронные операции ввода-вывода для чтения данных из файла *data.txt* с заданным размером буфера *bufferSize*. Она открывает файл, выделяет память под буфер и запускает цикл, инициирующий операции чтения с помощью функции *ReadFileEx*. Каждое чтение вызывает *CallbackWhenReadCompletion*, что позволяет выполнять обработку данных в фоновом режиме.

*«fileOperationsSync()»*: функция отвечает за синхронное чтение данных из файла data.txt. Она открывает файл, считывает все его содержимое в буфер и обрабатывает данные, сортируя их с помощью *insertionSort*. После сортировки отсортированные данные записываются в файл *stats\_sync.txt*.

*«main()»*: Главная функция программы, которая инициирует процесс обработки. Она вызывает функцию *generateFile* для создания исходного файла с данными, затем запускает асинхронные операции ввода-вывода, варьируя размер буфера для оценки производительности. В конце программа вызывает синхронную обработку данных и выводит результаты работы в консоль, включая время выполнения и производительность. Результат выполнения программы на рисунке 3.1.

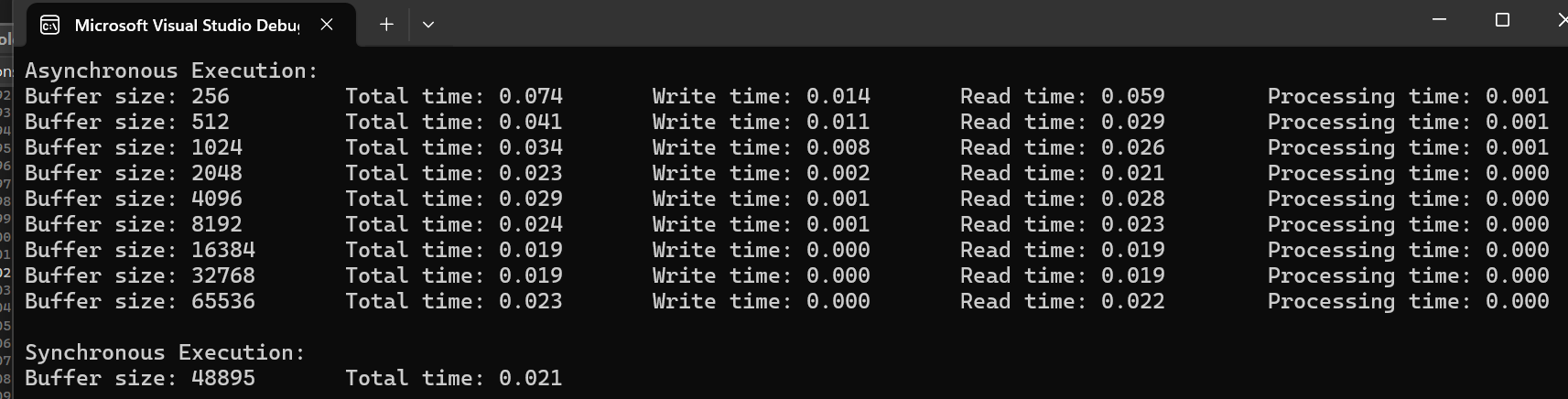


Рисунок 3.1 – Результат выполнения программы.

Результаты асинхронного выполнения показывают, что время, затраченное на операции ввода-вывода, значительно варьируется в зависимости от размера буфера, используемого для чтения данных:

1 меньшие буферы (256–2048 байт): временные затраты на чтение данных были более заметными. Например, при размере буфера 256 байт общее время выполнения составило 0.074 секунды, из которых на чтение ушло 0.059 секунды. Это указывает на то, что при использовании слишком маленького буфера происходит большая частота обращений к файловой системе;

2 средние буферы (4096–16384 байт): с увеличением размера буфера, общее время выполнения значительно уменьшается. При размере буфера 16384 байт общее время составило всего 0.019 секунды, с минимальными затратами на запись и чтение. Это указывает на более эффективное использование ресурсов и меньшую частоту операций ввода-вывода;

3 большие буферы (32768–65536 байт): дальнейшее увеличение размера буфера до 65536 байт приводит к незначительному увеличению общего времени выполнения до 0.023 секунды. Это может свидетельствовать о том, что оптимальный размер буфера был достигнут, и слишком большие буферы могут привести к избыточным затратам памяти без значительного выигрыша в производительности.

Асинхронные операции показали свою эффективность, особенно при использовании средних размеров буфера, что позволяет оптимально балансировать между временем выполнения и использованием ресурсов.

Результаты синхронного выполнения были представлены одним измерением, где размер буфера составил 48895 байт и общее время выполнения — 0.021 секунды.

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 <iomanip>

#include <chrono>

#include <cmath>

#include <vector>

#include <sstream>

#include <fstream>

#include <cstdlib>

#include <ctime>

#include <numeric>

#include <algorithm>

#define BASE\_BUFFER\_SIZE 256

using namespace std;

DWORD asyncIterationCount = 0;

DWORD syncIterationCount = 0;

int asyncReadCount = 0;

std::chrono::duration<double> asyncReadDuration;

std::chrono::duration<double> processingDuration;

std::chrono::duration<double> asyncWriteDuration;

void computeStatistics(const std::vector<int>& data, double& mean, double& stdev, int& minValue, int& maxValue) {

int size = data.size();

if (size == 0) return;

mean = std::accumulate(data.begin(), data.end(), 0.0) / size;

minValue = \*std::min\_element(data.begin(), data.end());

maxValue = \*std::max\_element(data.begin(), data.end());

double varianceSum = 0.0;

for (int value : data) {

varianceSum += (value - mean) \* (value - mean);

}

stdev = std::sqrt(varianceSum / size);

}

void createDataFile(const std::wstring& filename, int count) {

std::ofstream outFile(filename);

if (outFile.is\_open()) {

for (int i = 0; i < count; i++) {

outFile << count - i << " ";

}

outFile.close();

}

else {

std::wcout << L"Error opening file for writing: " << filename << "\n";

}

}

void saveToFile(double mean, double stdev, int minValue, int maxValue, const std::wstring& filename) {

HANDLE hFile = CreateFile(

filename.c\_str(),

GENERIC\_WRITE,

FILE\_APPEND\_DATA,

NULL,

OPEN\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL,

NULL

);

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

DWORD writtenBytes;

std::wstring result = L"Mean: " + std::to\_wstring(mean) + L", Std Dev: " + std::to\_wstring(stdev) +

L", Min: " + std::to\_wstring(minValue) + L", Max: " + std::to\_wstring(maxValue) + L"\n";

WriteFile(hFile, result.c\_str(), result.size() \* sizeof(wchar\_t), &writtenBytes, NULL);

CloseHandle(hFile);

}

else {

std::wcerr << L"Error opening file for writing: " << filename << '\n';

}

}

void CALLBACK onAsyncReadComplete(DWORD errorCode, DWORD bytesRead, LPOVERLAPPED lpOverlapped) {

if (errorCode == 0) {

lpOverlapped->Offset += bytesRead;

char\* buffer = (char\*)lpOverlapped->hEvent;

buffer[bytesRead] = '\0';

std::vector<int> data;

std::istringstream iss(buffer);

int value;

while (iss >> value) {

data.push\_back(value);

}

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

double mean, stdev;

int minValue, maxValue;

computeStatistics(data, mean, stdev, minValue, maxValue);

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

processingDuration = processingEnd - processingStart;

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

saveToFile(mean, stdev, minValue, maxValue, L"stats\_async.txt");

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

asyncWriteDuration = processingEnd - processingStart;

}

else {

if (errorCode == 38) {

return;

}

std::cerr << "Error reading file: " << errorCode << std::endl;

}

}

void performAsyncFileOperations(int bufferSize) {

HANDLE hFile = CreateFile(L"data.txt", GENERIC\_READ, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL | FILE\_FLAG\_OVERLAPPED, NULL);

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

std::cerr << "Error opening file: " << GetLastError() << std::endl;

return;

}

char\* buffer = new char[bufferSize];

OVERLAPPED overlapped = { 0 };

overlapped.hEvent = (HANDLE)buffer;

std::ofstream outFile(L"stats\_async.txt");

outFile.close();

while (true) {

DWORD previousOffset = overlapped.Offset;

if (!ReadFileEx(hFile, buffer, bufferSize - 1, &overlapped, onAsyncReadComplete)) {

std::cerr << "Error initiating read: " << GetLastError() << std::endl;

CloseHandle(hFile);

return;

}

asyncReadCount++;

SleepEx(INFINITE, TRUE);

auto fileSize = GetFileSize(hFile, NULL);

if (fileSize <= overlapped.Offset) {

break;

}

if (previousOffset == overlapped.Offset) {

break;

}

}

CloseHandle(hFile);

delete[] buffer;

}

DWORD performSyncFileOperations() {

HANDLE hFile = CreateFile(

L"data.txt",

GENERIC\_READ,

0,

NULL,

OPEN\_EXISTING,

FILE\_ATTRIBUTE\_NORMAL,

NULL);

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

std::cerr << "Could not open file (Error: " << GetLastError() << ").\n";

return -1;

}

DWORD fileSize = GetFileSize(hFile, NULL);

if (fileSize == INVALID\_FILE\_SIZE) {

std::cerr << "Could not get file size (Error: " << GetLastError() << ").\n";

CloseHandle(hFile);

return -1;

}

char\* buffer = new char[fileSize + 1];

if (!buffer) {

std::cerr << "Memory allocation failed." << std::endl;

CloseHandle(hFile);

return -1;

}

DWORD bytesRead;

if (!ReadFile(hFile, buffer, fileSize, &bytesRead, NULL)) {

std::cerr << "Could not read file (Error: " << GetLastError() << ")." << std::endl;

delete[] buffer;

CloseHandle(hFile);

return -1;

}

buffer[bytesRead] = '\0';

std::vector<int> data;

std::istringstream iss(buffer);

int value;

while (iss >> value) {

data.push\_back(value);

}

double mean, stdev;

int minValue, maxValue;

computeStatistics(data, mean, stdev, minValue, maxValue);

saveToFile(mean, stdev, minValue, maxValue, L"stats\_sync.txt");

delete[] buffer;

CloseHandle(hFile);

return fileSize + 1;

}

int main() {

createDataFile(L"data.txt", 10000);

int i = 0;

bool stop = false;

std::cout << "Asynchronous Execution:\n";

while (true) {

int bufferMultiplier = pow(2, i);

i++;

int maxBufferSize = BASE\_BUFFER\_SIZE \* bufferMultiplier;

if (maxBufferSize >= 1e5) {

break;

}

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

performAsyncFileOperations(maxBufferSize);

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

std::chrono::duration<double> asyncDuration = asyncEnd - asyncStart;

std::cout << "Buffer size: " << maxBufferSize << '\t'

<< fixed << setprecision(3) << " Total time: " << asyncDuration.count() << '\t'

<< " Write time: " << asyncWriteDuration.count() \* asyncReadCount << "\t"

<< " Read time: " << abs(asyncDuration.count() - (asyncWriteDuration.count() + processingDuration.count()) \* asyncReadCount) << "\t"

<< " Processing time: " << processingDuration.count() \* asyncReadCount << "\n";

asyncIterationCount = 0;

asyncReadCount = 0;

}

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

auto syncBufferSize = performSyncFileOperations();

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

std::chrono::duration<double> syncDuration = syncEnd - syncStart;

std::cout << "\nSynchronous Execution:\n";

std::cout << "Buffer size: " << syncBufferSize << '\t'

<< " Total time: " << syncDuration.count() << '\n';

return 0;

}