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   
Севрюков С.И.

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 6](#_Toc178641669)

[Список использованных источников 7](#_Toc178641670)

[Приложение А (обязательное) исходный код 8](#_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 использовании расширенных и дополнительных возможностей и механизмов работы с файлами и организации ввода-вывода: неблокирующие и асинхронные операции, мультиплексирование ввода-вывода, отображение файлов в память, мультиплексирование.

В рамках задачи необходимо разработать приложение, которое выполняет обработку содержимого файла, используя асинхронные операции, и оценку производительности.

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.

# **2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ**

Файловая система представляет собой структуру, организующую хранение данных и доступ к ним на различных носителях, таких как жесткие диски, флешки или сетевые хранилища. Она устанавливает правила именования и размещения файлов, а также определяет способы взаимодействия программ с данными. Кроме того, файловая система предоставляет *API* для работы с файлами, что позволяет приложениям открывать, читать, записывать и закрывать файлы.

Одним из важных аспектов взаимодействия программ с файлами и внешними устройствами является ввод-вывод, который может быть синхронным и асинхронным. При синхронном вводе-выводе выполнение программы приостанавливается до завершения операции, что может привести к блокировке потока при медленном выполнении задачи, например, при чтении больших файлов или сетевом взаимодействии. Этот подход удобен в простых сценариях, но снижает производительность в случае длительных операций ввода-вывода.

Асинхронный ввод-вывод, также называемый перекрывающимся, позволяет программе продолжать выполнение других задач, пока операция ввода-вывода выполняется в фоновом режиме. Для этого используется структура *OVERLAPPED*, позволяющая передавать данные и отслеживать состояние операции. Функции, такие как *ReadFileEx*() и *WriteFileEx*(), позволяют запускать операции чтения и записи без ожидания завершения, тем самым освобождая поток для других задач. Операционная система уведомляет приложение о завершении асинхронной операции.

Отображение файлов в память — еще один способ работы с файлами, который позволяет проецировать их содержимое в оперативную память, что ускоряет доступ к данным. Вместо последовательного чтения и записи данные можно обрабатывать напрямую как массив в памяти. Для этого используются функции *CreateFileMapping*() и *MapViewOfFile*().

# **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.

– *createDataFile*: создаёт текстовый файл с определённым количеством чисел, который будет использоваться для операций чтения.

– *calculateStats*: рассчитывает статистику (среднее, стандартное отклонение, минимальное и максимальное значения) для заданного набора данных.

– *saveToFile*: сохраняет результаты статистического анализа в файл, используя асинхронные или синхронные методы записи.

– *onAsyncReadComplete*: функция обратного вызова, обрабатывающая завершение асинхронной операции чтения. Она получает данные, обрабатывает их и сохраняет результаты в файл.

– *executeAsyncOperations*: выполняет асинхронное чтение из файла, увеличивая размер буфера для каждого шага и измеряя время на чтение, запись и обработку данных.

– execute*SyncOperations*: выполняет синхронное чтение из файла и сохраняет статистику в отдельный файл.

Результат работы программы на рисунке 3.1.

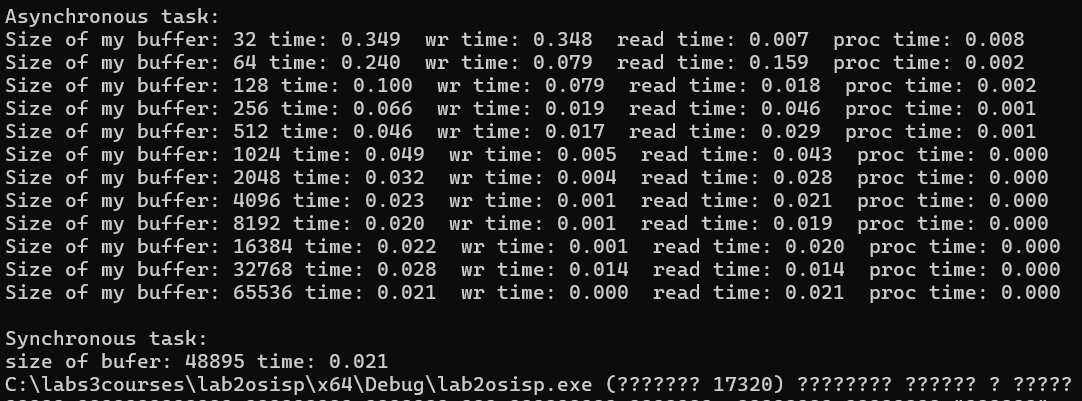


Рисунок 3.1 – Результат программы

Результаты асинхронного выполнения показывают, что время, затраченное на операции ввода-вывода, значительно варьируется в зависимости от размера буфера, используемого для чтения данных, и более оптимальны при средних размерах буфера.

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

# **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.

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.

Меньший буфер данных не означает меньшее время выполнения задачи, а наоборот, так как при использовании слишком маленького буфера происходит большая частота обращений к файловой системе.

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**

#include <windows.h>

#include <fstream>

#include <vector>

#include <sstream>

#include <numeric>

#include <algorithm>

#include <chrono>

#include <cmath>

#include <iostream>

#define INITIAL\_BUFFER\_SIZE 256

using namespace std;

// Глобальные переменные для статистики

DWORD asyncIterations = 0;

DWORD syncIterations = 0;

int totalAsyncReads = 0;

chrono::duration<double> asyncReadTime, processingTime, asyncWriteTime;

void calculateStats(const vector<int>& numbers, double& avg, double& deviation, int& minimum, int& maximum) {

if (numbers.empty()) return;

int count = numbers.size();

avg = accumulate(numbers.begin(), numbers.end(), 0.0) / count;

minimum = \*min\_element(numbers.begin(), numbers.end());

maximum = \*max\_element(numbers.begin(), numbers.end());

double varianceSum = 0.0;

for (int num : numbers) {

varianceSum += pow(num - avg, 2);

}

deviation = sqrt(varianceSum / count);

}

void generateFileData(const wstring& filePath, int numValues) {

ofstream outputFile(filePath);

if (outputFile.is\_open()) {

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

outputFile << (numValues - i) << " ";

}

outputFile.close();

}

}

void saveStatsToFile(double avg, double deviation, int minimum, int maximum, const wstring& filePath) {

HANDLE fileHandle = CreateFile(

filePath.c\_str(),

GENERIC\_WRITE,

FILE\_APPEND\_DATA,

nullptr,

OPEN\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL,

nullptr

);

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

DWORD writtenBytes;

wstring statsString = L"Mean: " + to\_wstring(avg) + L", Std Dev: " + to\_wstring(deviation) +

L", Min: " + to\_wstring(minimum) + L", Max: " + to\_wstring(maximum) + L"\n";

WriteFile(fileHandle, statsString.c\_str(), statsString.size() \* sizeof(wchar\_t), &writtenBytes, nullptr);

CloseHandle(fileHandle);

}

}

void CALLBACK asyncReadComplete(DWORD error, DWORD bytesRead, LPOVERLAPPED overlapped) {

if (error == 0) {

overlapped->Offset += bytesRead;

char\* buffer = reinterpret\_cast<char\*>(overlapped->hEvent);

buffer[bytesRead] = '\0';

vector<int> values;

istringstream dataStream(buffer);

int number;

while (dataStream >> number) {

values.push\_back(number);

}

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

double mean, stdev;

int minValue, maxValue;

calculateStats(values, mean, stdev, minValue, maxValue);

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

processingTime = endProcessing - startProcessing;

startProcessing = chrono::high\_resolution\_clock::now();

saveStatsToFile(mean, stdev, minValue, maxValue, L"async\_statistics.txt");

endProcessing = chrono::high\_resolution\_clock::now();

asyncWriteTime = endProcessing - startProcessing;

}

}

void executeAsyncOperations(int bufferCapacity) {

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

if (fileHandle == INVALID\_HANDLE\_VALUE) return;

char\* buffer = new char[bufferCapacity];

OVERLAPPED overlappedRegion = { 0 };

overlappedRegion.hEvent = reinterpret\_cast<HANDLE>(buffer);

ofstream resetFile(L"async\_statistics.txt");

resetFile.close();

while (true) {

DWORD previousOffset = overlappedRegion.Offset;

if (!ReadFileEx(fileHandle, buffer, bufferCapacity - 1, &overlappedRegion, asyncReadComplete)) {

CloseHandle(fileHandle);

delete[] buffer;

return;

}

totalAsyncReads++;

SleepEx(INFINITE, TRUE);

DWORD fileSize = GetFileSize(fileHandle, nullptr);

if (fileSize <= overlappedRegion.Offset || previousOffset == overlappedRegion.Offset) {

break;

}

}

CloseHandle(fileHandle);

delete[] buffer;

}

DWORD executeSyncOperations() {

HANDLE fileHandle = CreateFile(L"data\_sync.txt", GENERIC\_READ, 0, nullptr, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, nullptr);

if (fileHandle == INVALID\_HANDLE\_VALUE) return -1;

DWORD fileSize = GetFileSize(fileHandle, nullptr);

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

CloseHandle(fileHandle);

return -1;

}

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

DWORD bytesRead;

if (!ReadFile(fileHandle, buffer, fileSize, &bytesRead, nullptr)) {

delete[] buffer;

CloseHandle(fileHandle);

return -1;

}

buffer[bytesRead] = '\0';

vector<int> numbers;

istringstream inputStream(buffer);

int num;

while (inputStream >> num) {

numbers.push\_back(num);

}

double avg, stdev;

int min, max;

calculateStats(numbers, avg, stdev, min, max);

saveStatsToFile(avg, stdev, min, max, L"sync\_statistics.txt");

delete[] buffer;

CloseHandle(fileHandle);

return fileSize;

}

int main() {

generateFileData(L"data\_async.txt", 10000);

generateFileData(L"data\_sync.txt", 10000);

cout << "Asynchronous Processing:\n";

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

int bufferFactor = static\_cast<int>(pow(2, i));

int bufferSize = INITIAL\_BUFFER\_SIZE \* bufferFactor;

if (bufferSize >= 100000) break;

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

executeAsyncOperations(bufferSize);

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

chrono::duration<double> totalTime = endTime - startTime;

cout << "Buffer size: " << bufferSize << " | Total Time: " << fixed << setprecision(3) << totalTime.count() << "s\n";

totalAsyncReads = 0;

}

cout << "\nSynchronous Processing:\n";

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

DWORD syncSize = executeSyncOperations();

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

chrono::duration<double> syncTime = endSync - startSync;

cout << "Buffer size: " << syncSize << " | Total Time: " << syncTime.count() << "s\n";

return 0;

}