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.253505 Павлович В.Ю.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc178979879)

[2 Краткие теоретические сведения 4](#_Toc178979880)

[3 Описание работы программы 5](#_Toc178979881)

[3.1 Традиционная обработка файла 5](#_Toc178979882)

[3.2 Асинхронная обработка файла 5](#_Toc178979883)

[3.3 Измерение времени исполнения 5](#_Toc178979884)

[Заключение 7](#_Toc178979885)

[Список использованных источников 8](#_Toc178979886)

[Приложение А (Обязательное) Исходный код программы 9](#_Toc178979887)

# **1 ПОСТАНОВКА ЗАДАЧИ**

Целью лабораторной работы является изучение расширенных/дополнительных возможностей и механизмов работы с файлами и организации ввода-вывода: неблокирующие и асинхронные операции, мультиплексирование ввода-вывода, отображение файлов в память, мультиплексирование и др.

В качестве задачи необходимо разработать приложение, обеспечивающее реализацию обработки содержимого файла данных, используя асинхронный ввод-вывод (чтение/запись очередных порций данных параллельно с выполнением обработки данных в памяти).

Необходимо произвести поиск «узкого места» в программной реализации. Оценку эффективности (производительности) по сравнению с традиционным подходом (чтение → обработка → выгрузка).

В качестве операции, проводимой над файлом, является подсчет количества латинских букв в текстовом файле.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Существует два типа синхронизации ввода/вывода (I/O): синхронный и асинхронный ввод/вывод. Асинхронный ввод/вывод также называют перекрывающимся вводом/выводом.

При синхронном файловом вводе/выводе поток начинает операцию ввода/вывода и сразу же переходит в состояние ожидания, пока запрос ввода/вывода не будет завершен. Поток, выполняющий асинхронный файловый ввод/вывод, отправляет запрос ввода/вывода ядру, вызывая соответствующую функцию. Если запрос принят ядром, вызывающий поток продолжает обрабатывать другую задачу, пока ядро не сигнализирует потоку о завершении операции ввода/вывода. Затем он прерывает текущую задачу и обрабатывает данные из операции ввода/вывода по мере необходимости.

В ситуациях, когда запрос ввода/вывода ожидается длительным, например, обновление или резервное копирование большой базы данных или медленное соединение, асинхронный ввод/вывод является хорошим способом оптимизации эффективности обработки. Однако для относительно быстрых операций ввода/вывода накладные расходы на обработку запросов ввода/вывода ядра и сигналов ядра могут сделать асинхронный ввод/вывод менее выгодным, особенно если необходимо выполнить множество быстрых операций ввода/вывода. В этом случае синхронный ввод/вывод будет лучше. Механизмы и детали реализации этих задач зависят от типа используемого дескриптора устройства и конкретных нужд приложения. Другими словами, обычно существует несколько способов решения проблемы.[1]

3 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

В этом разделе будут рассмотрены основные функции программы, которые были задействованы для выполнения работы. Для сравнения скорости работы для всех операций использовался один и тот же текстовый документ, состоящий из 109 символов, в котором подсчитывается количество латинских букв.

## **3.1 Традиционная обработка файла**

Функция process выполняет традиционную обработку файла, а именно полную выгрузку содержимого файла в оперативную память с последующей обработкой его содержимого в оперативной памяти. Обработка файла заключается в подсчете количества латинских букв в файле путем проверки каждого символа файла.

## **3.2 Асинхронная обработка файла**

Функция processAsync выполняет асинхронную обработку содержимого файла. В качестве параметра она принимает размер буфера для чтения данных файла. Внутри функции происходит чтение блока данных размером с буфер из файла, после чего данные в буфере обрабатываются той же функцией, что и при традиционной обработке. Во время обработки буфера запускается асинхронное чтение следующего блока данных. Данная операция повторяется до завершения файла. Завершение файла определяется получением кода ошибки 38 ERROR\_HANDLE\_EOF [2]. Внутри данной функции используется процедура ReadFileEx и callback функция ReadCompletionRoutine, вызывающаяся по завершению асинхронного чтения содержимого файла.

## **3.3 Измерение времени исполнения**

Время выполнения процедур можно эффективно измерять с помощью объектов класса std::chrono::high\_resolution\_clock, который предоставляет высокую точность для временных замеров. Для этого необходимо зафиксировать момент старта выполнения процедуры, а затем зафиксировать момент завершения.

Процесс измерения времени работы выглядит следующим образом: в момент начала процедуры вызывается метод now() объекта high\_resolution\_clock, который возвращает текущее время. После завершения процедуры снова вызывается метод now(), чтобы получить время завершения. Разница между этими двумя временными отметками позволяет определить общее время выполнения процедуры.

Для удобства работы с полученными временными значениями используется объект класса std::chrono::duration<double>. Этот объект представляет собой интервал времени, который можно легко вычислить, вычитая время начала из времени завершения.

В процессе замеров проводилось сравнение времени выполнения асинхронных процедур с различными размерами буфера с традиционной обработкой файла. Результат работы программы представлен на рисунке 3.1.

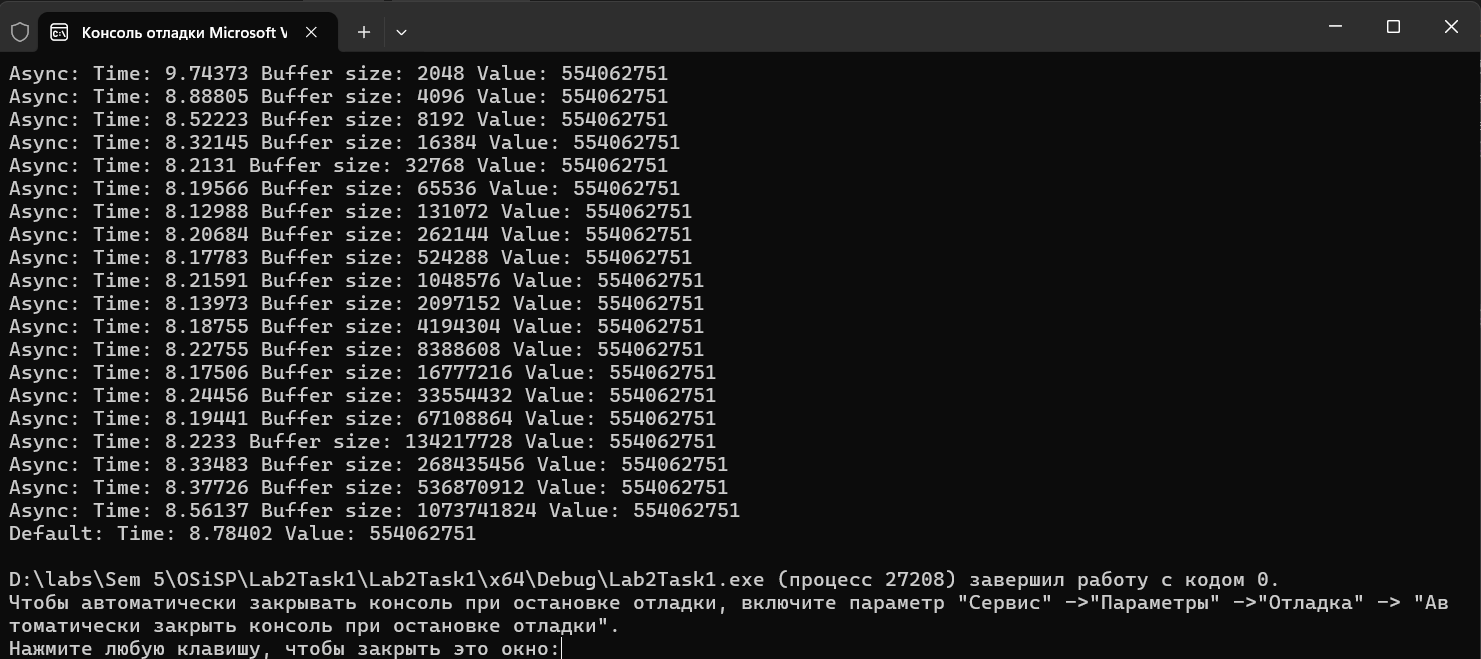


Рисунок 3.1 – Результат работы программы

В выходных данных описан тип операции (асинхронная/традиционная), время работы данной операции и количество букв в обрабатываемом файле. Для асинхронных операций также указан размер буфера в байтах.

# **ЗАКЛЮЧЕНИЕ**

В ходе выполнения лабораторной работы было произведено сравнение производительности традиционного подхода к обработке файла (чтение → обработка → выгрузка) с асинхронным подходом, а также было произведено сравнение производительности асинхронной обработки с различным размером буфера.

Асинхронная реализация при правильном выборе размера буфера показала заметно лучшие результаты, чем традиционная, при этом некорректный размер буфера (слишком маленький или слишком большой) негативно сказывается на производительности асинхронной обработки вплоть до того, что данная реализация оказывается медленнее традиционной.

Таким образом, сравнительный анализ показал, что асинхронная обработка файлов позволяет увеличить работоспособность программы, но только при условии корректного выбора размера блока считываемых за раз данных для конкретного объема файла.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Синхронный и асинхронный ввод-вывод – Win32 apps [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows/win32/fileio/synchronous-and-asynchronous-i-o>.

[2] Коды системных ошибок (0–499) (WinError.h) – Win32 apps [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/windows/win32/debug/system-error-codes--0-499->.

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код программы**

#include <windows.h>

#include <iostream>

#include <chrono>

#include <cmath>

#define MIN\_BUFFER\_SIZE 2048

int lettersAsync = 0;

int letters = 0;

void CALLBACK ReadCompletionRoutine(DWORD dwErrorCode, DWORD dwNumberOfBytesTransferred, LPOVERLAPPED lpOverlapped) {

if (dwErrorCode == 0) {

lpOverlapped->Offset += dwNumberOfBytesTransferred;

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

buffer[dwNumberOfBytesTransferred] = '\0';

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

if ((buffer[i] >= 65 && buffer[i] <= 90) || (buffer[i] >= 97 && buffer[i] <= 122)) {

lettersAsync++;

}

}

}

else {

if (dwErrorCode == 38) {

return;

}

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

}

}

void processAsync(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;

while (true) {

DWORD prev\_offset = overlapped.Offset;

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

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

CloseHandle(hFile);

return;

}

SleepEx(INFINITE, TRUE);

if (prev\_offset == overlapped.Offset) {

break;

}

}

CloseHandle(hFile);

delete[] buffer;

}

void process() {

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() << ")." << std::endl;

return;

}

DWORD fileSize = GetFileSize(hFile, NULL);

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

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

CloseHandle(hFile);

return;

}

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

if (!buffer) {

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

CloseHandle(hFile);

return;

}

DWORD bytesRead;

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

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

delete[] buffer;

CloseHandle(hFile);

return;

}

buffer[bytesRead] = '\0';

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

if ((buffer[i] >= 65 && buffer[i] <= 90) || (buffer[i] >= 97 && buffer[i] <= 122)) {

letters++;

}

}

delete[] buffer;

CloseHandle(hFile);

}

int main() {

int i = 0;

bool exit = false;

while(!exit) {

int buff\_exp = pow(2, i);

i++;

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

if (MIN\_BUFFER\_SIZE \* buff\_exp >= 1e9) {

exit = true;

}

processAsync(MIN\_BUFFER\_SIZE \* buff\_exp);

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

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

std::cout << "Async: Time: " << durationAsync.count() << " " << "Buffer size: " << MIN\_BUFFER\_SIZE \* buff\_exp << " " << "Value: " << lettersAsync << "\n";

lettersAsync = 0;

}

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

process();

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

std::chrono::duration<double> duration = end - start;

std::cout << "Default: Time: " << duration.count() << " " << "Value: " << letters << "\n";

return 0;

}