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 Снежко М.А.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

ОГЛАВЛЕНИЕ

[1 Формулировка задачи 3](#_Toc179742261)

[2 Краткие теоритические сведения 4](#_Toc179742262)

[3 Описание функций программы 5](#_Toc179742263)

[Заключение 6](#_Toc179742264)

[Список использованных источников 7](#_Toc179742265)

[Приложение А (обязательное) 8](#_Toc179742266)

# 1 ФОРМУЛИРОВКА ЗАДАЧИ

Цель лабораторной работы заключается в изучении механизмов расширенной работы с файлами и организации ввода-вывода. В процессе выполнения лабораторной работы изучаются такие темы, как неблокирующие и асинхронные операции, мультиплексирование ввода-вывода и отображение файлов в память. Задачей лабораторной работы является разработка программы на языке C++ для обработки данных из файла с применением механизма отображения файлов в память (memory-mapped files).

Особое внимание уделяется оценке производительности при многопоточном выполнении сортировки, что позволяет сравнить этот подход с традиционным методом последовательного чтения, обработки и записи данных. Важной особенностью является отсутствие мер для предотвращения коллизий при параллельном доступе, что дает возможность исследовать, как многопоточный доступ влияет на общую производительность программы.

Таким образом, целью данной работы является выявление преимуществ и ограничений использования многопоточности в условиях обработки больших объемов данных, а также оценка эффективности использования memory-mapped files в современных вычислительных системах. В лабораторной работе также рассматриваются возможные варианты обработки данных, такие как сортировка, статистический анализ и криптография, причем объем данных должен быть достаточно большим для проведения заметных и измеримых расчетов. Проводится оценка эффективности подхода по сравнению с традиционным способом (последовательное чтение, обработка и запись), включая многопоточные версии. Для корректной оценки результатов необходимо выбирать объем данных, который обеспечит достаточную длительность обработки, чтобы производительность можно было заметно измерить.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Подсистема ввода-вывода обеспечивает связь вычислительной системы с внешними (по отношению к ЦП и памяти) узлами и устройствами.

Все программы и данные хранятся в долговременной (внешней) памяти компьютера в виде файлов. Файл – это определенное количество информации (программа или данные), имеющее имя и хранящееся в долговременной (внешней) памяти.

Имя файла состоит из двух частей, разделенных точкой: имя файла и расширение, определяющее его тип. Имя файлу дает пользователь, а тип файла обычно задается программой автоматически при его создании.

На каждом носителе информации (гибком, жестком или лазерном диске) может храниться большое количество файлов. Порядок хранения файлов на диске определяется установленной файловой системой.

Файловая система – это система хранения файлов и организации каталогов.

Объект синхронизации – это объект, дескриптор которого можно указать в одной из функций ожидания для координации выполнения нескольких потоков. Несколько процессов могут иметь дескриптор одного и того же объекта синхронизации, что делает возможной синхронизацию между процессами. [1]

Отображение файла в память – это способ работы с файлами в некоторых операционных системах, при котором всему файлу или некоторой непрерывной его части ставится в соответствие определённый участок памяти (диапазон адресов оперативной памяти). При этом чтение данных из этих адресов фактически приводит к чтению данных из отображенного файла, а запись данных по этим адресам приводит к записи этих данных в файл. Отображать на память часто можно не только обычные файлы, но и файлы устройств. [2]

Мультиплексирование ввода-вывода – это техника, которая позволяет управлять несколькими потоками ввода-вывода с помощью одного потока управления. С помощью функций система отслеживает состояние нескольких файловых дескрипторов, определяя, какой из них готов к операциям ввода-вывода. Эти подходы обеспечивают более эффективное управление ресурсами и позволяют обрабатывать множество потоков ввода-вывода без создания большого количества потоков.

Хотя неблокирующие и асинхронные операции, а также мультиплексирование ввода-вывода довольно значительно повышают производительность приложений, работающих с вводом-выводом, их реализация может быть более сложной из-за необходимости управления состоянием и синхронизацией. Тем не менее, эти техники являются важными инструментами для оптимизации работы с большими объемами данных и обеспечения высокой отзывчивости приложений. [3]

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Исходя из формулировки задачи, была разработана программа, реализующая обработку содержимого файла данных, используя отображение в память, а также сравнения по скорости выполнения между отображением в память и традиционным способов. Результат работы программы представлен на рисунке 3.1.

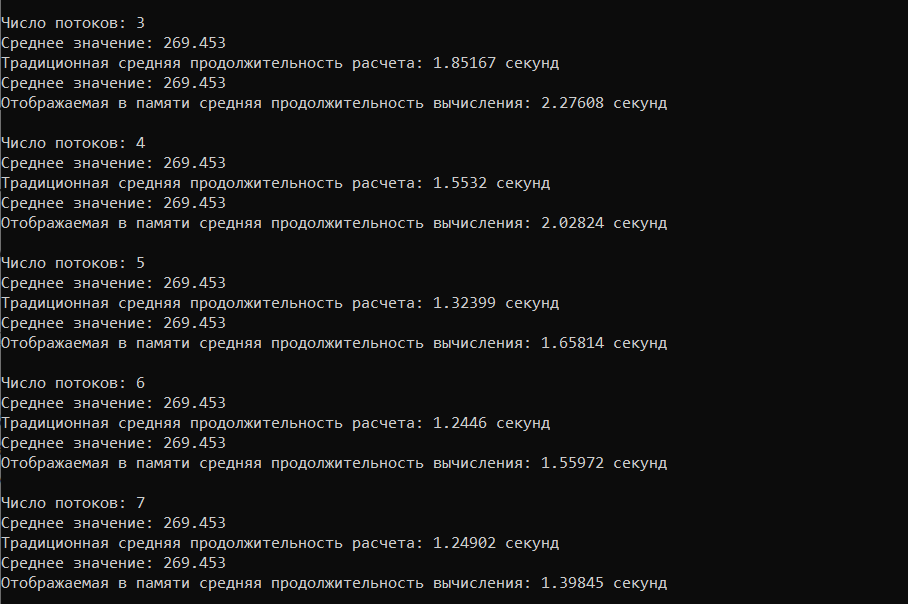


Рисунок 3.1 – Результат работы программы

Таким образом, после нескольких запусков программы можно сделать вывод о том, что скорости выполнения близки, что говорит о незначительной разнице в скорости рассматриваемых подходов.

# ЗАКЛЮЧЕНИЕ

В заключение данной лабораторной работы по дисциплине «Операционные системы и системное программирование» можно отметить, что разработанная программа для обработки файлов с использованием механизма отображения в память позволила на практике исследовать эффективность различных методов ввода-вывода. Основной акцент был сделан на сравнении производительности при многопоточной обработке данных, что дало возможность продемонстрировать реальные преимущества и ограничения работы в параллельных потоках.

Процесс разработки включал в себя тщательное изучение системных вызовов и API для работы с потоками и файлами, что позволило более глубоко понять структуру и особенности современных файловых систем. Применение многопоточного программирования, особенно без использования механизмов для предотвращения коллизий, позволило проанализировать, как параллельный доступ к данным может влиять на общую производительность.

Полученные результаты демонстрируют важность правильного выбора методов ввода-вывода, особенно при работе с большими объемами данных. В ходе работы было показано, что применение отображения файлов в память значительно ускоряет обработку данных по сравнению с традиционными методами. Кроме того, варьирование количества потоков подтвердило важность грамотной организации многопоточной обработки для достижения максимальной производительности.

Таким образом, лабораторная работа помогла не только закрепить теоретические знания, но и получить практические навыки, которые будут полезны в дальнейшем при реализации сложных системных проектов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Microsoft "Synchronization objects" – Электронный ресурс. Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/sync/synchronization-objects

[2] StudFiles – Электронный ресурс. – Режим доступа: https://studfile.net/

[3] Microsoft. "File Management" – Электронный ресурс. – Режим доступа: https://docs.microsoft.com/en-us/windows/win32/fileio/file-management.

ПРИЛОЖЕНИЕ А

(обязательное)

**Исходный код программы**

#include <iostream>

#include <vector>

#include <cmath>

#include <chrono>

#include <windows.h>

#include <cstdlib>

const size\_t MAX\_NUM\_THREADS = 16;

const size\_t NUM\_INTS = 1024 \* 1024 \* 128;

size\_t NUM\_THREADS;

struct ThreadDataMMap

{

int\* start;

size\_t length;

long long partialSum;

};

struct ThreadDataTraditional

{

int\* start;

size\_t length;

long long partialSum;

HANDLE file;

size\_t offset;

};

int powi(int x, int power)

{

int a = 1;

for (int i = 0; i < power; i++)

{

a \*= x;

}

return a;

}

DWORD WINAPI calculateSumMMap(LPVOID param)

{

ThreadDataMMap\* data = static\_cast<ThreadDataMMap\*>(param);

for (int\* current = data->start; current < data->start + data->length; current++)

{

\*current = powi(\*current, 3) - powi(\*current, 2) + powi(\*current, 1);

data->partialSum += \*current;

}

return 0;

}

DWORD WINAPI calculateSumTraditional(LPVOID param)

{

ThreadDataTraditional\* data = static\_cast<ThreadDataTraditional\*>(param);

for (int\* current = data->start; current < data->start + data->length; current++)

{

\*current = powi(\*current, 3) - powi(\*current, 2) + powi(\*current, 1);

data->partialSum += \*current;

}

DWORD bytesWritten;

SetFilePointer(data->file, data->offset \* sizeof(int), nullptr, FILE\_BEGIN);

WriteFile(data->file, data->start, data->length \* sizeof(int), &bytesWritten, nullptr);

return 0;

}

void fillFileWithInts(const char\* filename, size\_t numInts)

{

HANDLE file = CreateFileA(

filename,

GENERIC\_WRITE,

0,

nullptr,

CREATE\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL,

nullptr);

std::vector<int> data(numInts);

for (size\_t i = 0; i < numInts; i++)

{

data[i] = rand() % 10 + 1;

}

DWORD bytesWritten;

WriteFile(file, data.data(), numInts \* sizeof(int), &bytesWritten, nullptr);

CloseHandle(file);

}

void mmapAverage(const char\* filename)

{

HANDLE file = CreateFileA(

filename,

GENERIC\_READ | GENERIC\_WRITE,

0,

nullptr,

OPEN\_EXISTING,

FILE\_ATTRIBUTE\_NORMAL,

nullptr);

LARGE\_INTEGER fileSize;

GetFileSizeEx(file, &fileSize);

HANDLE fileMapping = CreateFileMappingA(

file,

nullptr,

PAGE\_READWRITE,

0,

0,

nullptr);

int\* data = static\_cast<int\*>(MapViewOfFile(

fileMapping,

FILE\_MAP\_ALL\_ACCESS,

0,

0,

0));

size\_t numInts = fileSize.QuadPart / sizeof(int);

size\_t chunkSize = numInts / NUM\_THREADS;

std::vector<HANDLE> threads(NUM\_THREADS);

std::vector<ThreadDataMMap> threadData(NUM\_THREADS);

for (size\_t i = 0; i < NUM\_THREADS; i++)

{

size\_t start = i \* chunkSize;

size\_t end = (i == NUM\_THREADS - 1) ? (numInts) : (start + chunkSize);

threadData[i] = { data + start, end - start, 0 };

threads[i] = CreateThread(

nullptr,

0,

calculateSumMMap,

&threadData[i],

0,

nullptr);

}

WaitForMultipleObjects(NUM\_THREADS, threads.data(), TRUE, INFINITE);

long double totalSum = 0;

for (size\_t i = 0; i < NUM\_THREADS; ++i)

{

totalSum += threadData[i].partialSum;

CloseHandle(threads[i]);

}

long double totalAverage = totalSum / numInts;

std::cout << "Среднее значение: " << totalAverage << '\n';

UnmapViewOfFile(data);

CloseHandle(fileMapping);

CloseHandle(file);

}

void traditionalAverage(const char\* filename)

{

HANDLE file = CreateFileA(

filename,

GENERIC\_READ,

0,

nullptr,

OPEN\_EXISTING,

FILE\_ATTRIBUTE\_NORMAL,

nullptr);

LARGE\_INTEGER fileSize;

GetFileSizeEx(file, &fileSize);

size\_t numInts = fileSize.QuadPart / sizeof(int);

std::vector<int> data(numInts);

DWORD bytesRead;

ReadFile(file, data.data(), fileSize.QuadPart, &bytesRead, nullptr);

size\_t chunkSize = numInts / NUM\_THREADS;

std::vector<HANDLE> threads(NUM\_THREADS);

std::vector<ThreadDataTraditional> threadData(NUM\_THREADS);

for (size\_t i = 0; i < NUM\_THREADS; i++)

{

size\_t start = i \* chunkSize;

size\_t end = (i == NUM\_THREADS - 1) ? numInts : start + chunkSize;

threadData[i] = { data.data() + start, end - start, 0, file, start };

threads[i] = CreateThread(

nullptr,

0,

calculateSumTraditional,

&threadData[i],

0,

nullptr);

}

WaitForMultipleObjects(NUM\_THREADS, threads.data(), TRUE, INFINITE);

long double totalSum = 0;

for (size\_t i = 0; i < NUM\_THREADS; i++)

{

totalSum += threadData[i].partialSum;

CloseHandle(threads[i]);

}

long double totalAverage = totalSum / numInts;

std::cout << "Среднее значение: " << totalAverage << std::endl;

CloseHandle(file);

}

int main()

{

setlocale(LC\_ALL, "ru");

time\_t startTime = time(0);

const char\* filename = "largefile.dat";

for (NUM\_THREADS = 1; NUM\_THREADS <= MAX\_NUM\_THREADS; NUM\_THREADS++)

{

std::cout << "Число потоков: " << NUM\_THREADS << '\n';

srand(startTime);

fillFileWithInts(filename, NUM\_INTS);

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

traditionalAverage(filename);

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

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

std::cout << "Традиционная средняя продолжительность расчета: " << traditionalDurational.count() << " секунд\n";

srand(startTime);

fillFileWithInts(filename, NUM\_INTS);

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

mmapAverage(filename);

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

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

std::cout << "Отображаемая в памяти средняя продолжительность вычисления: " << mmapDuration.count() << " секунд\n";

std::cout << '\n';

}

return 0;

}