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

Фроленко К.Ю.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

СОДЕРЖАНИЕ

[1 Формулировка задачи 3](#_Toc178067889)

[2 Краткие теоритеческие сведения 4](#_Toc178067890)

[3 Описание функций программы 5](#_Toc178067891)

[3.1 Класс Database 5](#_Toc178067892)

[3.2 Класс SynchronousDatabase 5](#_Toc178067893)

[3.3 Функция createRecord 6](#_Toc178067894)

[3.4 Функция main 6](#_Toc178067895)

[4 Пример выполнения программы 7](#_Toc178067896)

[4.1 Запуск программы и процесс выполнения 7](#_Toc178067897)

[4.2 Описание работы и результатов 7](#_Toc178067898)

[Вывод 9](#_Toc178067899)

[Список использованных источников 10](#_Toc178067900)

[Приложение А (обязательное) 11](#_Toc178067901)

1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью моей лабораторной работы является разработка и анализ приложения, которое демонстрирует использование асинхронных методов ввода-вывода и технологии отображения файлов в память для обработки данных в операционной системе Windows. Эта задача актуальна для улучшения понимания влияния различных подходов к вводу-выводу на производительность системы. Цель работы заключается в том, чтобы исследовать и сравнить производительность асинхронных и синхронных методов, а также понять преимущества использования памяти в качестве промежуточного буфера для доступа к файлам.

В рамках лабораторной работы предполагается выполнение следующих задач:

1 Реализация асинхронного ввода-вывода: на первом этапе необходимо создать и запустить асинхронные операции чтения и записи данных с использованием соответствующих функций Windows API. Этот этап позволит оценить преимущества асинхронного доступа к данным по сравнению с традиционным синхронным вводом-выводом.

2 Отображение файла в память: следующим шагом будет настройка механизма отображения файла в оперативную память, что позволит обеспечить мгновенный доступ к данным и ускорить их обработку. Этот метод будет анализироваться на предмет его эффективности по сравнению с обычными подходами к чтению и записи файлов.

3 Анализ производительности: основной задачей будет сравнение времени выполнения операций чтения и записи при использовании различных методов ввода-вывода. Особое внимание будет уделено измерению и сравнению производительности асинхронного ввода-вывода и методов с отображением файла в память, чтобы определить оптимальные условия для их применения.

Этапы работы позволят не только углубить знания в области системного программирования, но и получить практический опыт в оптимизации процессов ввода-вывода, что способствует разработке более эффективных и производительных приложений.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

В современных операционных системах, таких как Windows, ввод-вывод файлов может быть реализован через различные механизмы, каждый из которых предоставляет уникальные преимущества в зависимости от приложения. Основные методы включают синхронный, асинхронный и неблокирующий ввод-вывод, а также мультиплексирование и отображение файлов в память.

Синхронный ввод-вывод является базовым методом, при котором операционная система приостанавливает выполнение вызывающего потока до завершения операции с файлом. Это обеспечивает упрощение программирования, но может снижать общую производительность приложения при обработке больших объемов данных или выполнении операций с множеством файлов.

Асинхронный ввод-вывод, реализуемый через структуру OVERLAPPED в Windows, позволяет приложению инициировать одну или несколько операций ввода-вывода, которые выполняются параллельно основному потоку выполнения. Это позволяет приложению продолжать работу, пока система обрабатывает ввод или вывод данных. Результаты асинхронных операций могут быть получены с помощью функций ожидания, таких как WaitForSingleObject.

Мультиплексирование ввода-вывода использует системные вызовы, такие как select() и poll(), для одновременного мониторинга нескольких источников ввода-вывода. Это позволяет программе реагировать на доступность данных для чтения или возможность записи без постоянного опроса каждого источника.

Отображение файлов в память предоставляет программам возможность работать с файлами так, как если бы они были частью адресного пространства приложения. Это достигается через вызовы CreateFileMapping() и MapViewOfFile(), которые создают сегмент памяти, отображаемый непосредственно на файл. Этот метод может значительно ускорить работу с файлами, особенно при частом доступе к случайным частям больших файлов.[1]

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

В рамках разработки программы для демонстрации работы с файловой базой данных были реализованы следующие функции:

– Класс Database;

– Класс SynchronousDatabase;

– Функция createRecord;

– main.

3.1 Класс Database

Класс Database представляет собой основу для работы с файловой базой данных с использованием отображения файла в память:

1 Конструктор Database – инициализирует базу данных, открывая файл для доступа и создания. После открытия файла, устанавливает его размер, соответствующий объему данных, рассчитанному исходя из количества записей, и создаёт отображение этого файла в память. Это отображение позволяет работать с файлом как с частью адресного пространства процесса, упрощая операции чтения и записи;

2 Деструктор ~Database – закрывает все открытые ресурсы, ассоциированные с базой данных, включая отображение файла в память и файл. Это обеспечивает корректное освобождение системных ресурсов и предотвращает утечки памяти, а также гарантирует сохранность данных при закрытии приложения;

3 writeRecord – выполняет запись данных в файловую базу данных. Для этого функция рассчитывает адрес в памяти, основываясь на индексе записи, и копирует данные непосредственно в этот адрес, используя отображение файла в память. Такой подход минимизирует задержки, связанные с вводом-выводом, и увеличивает скорость записи;

4 readRecord – считывает данные из базы данных, используя отображение файла в память. Адрес для чтения данных также рассчитывается исходя из индекса записи, что позволяет мгновенно получить доступ к необходимой информации без физического чтения диска;

3.2 Класс SynchronousDatabase

Класс SynchronousDatabase используется для демонстрации синхронного доступа к файловой базе данных:

1 Конструктор SynchronousDatabase – инициализирует синхронную базу данных, открывая файл для синхронного доступа. Устанавливает размер файла в соответствии с количеством записей и гарантирует его создание, если файл не существует;

2 Деструктор ~SynchronousDatabase – закрывает файл, управляя освобождением всех системных ресурсов, связанных с файлом, для предотвращения утечек памяти и нарушения целостности данных;

3 writeRecord – выполняет запись данных в файл, используя синхронные операции ввода-вывода. Функция устанавливает позицию в файле на основе индекса записи, что позволяет точно адресовать место записи в файле, и затем записывает данные, гарантируя их корректное сохранение;

4 readRecord – читает данные из файла, также используя синхронные операции ввода-вывода. Функция настраивает позицию чтения в файле согласно индексу записи и считывает данные, обеспечивая их точное извлечение для дальнейшей обработки или отображения.

3.3 Функция createRecord

Функция createRecord предназначена для создания новой записи типа Record, которая может быть использована в последующих операциях ввода-вывода. Функция совершает следующие действия:

1 Инициализация структуры Record: процесс включает в себя задание идентификатора и имени для записи. Идентификатор и имя инициализируются на основе передаваемых параметров;

2 Заполнениеполей структуры: функция устанавливает значения полей структуры, обеспечивая корректные данные для дальнейшей записи в файл или чтения из файла.

3.4 Функция main

Функция main является входной точкой программы и демонстрирует использование классов Database и SynchronousDatabase. Функция совершает следующие действия:

1 Инициализация: на этапе инициализации создаётся экземпляр класса Database;

2 Тестирование операций: запускаются операции записи и чтения через асинхронный и синхронный доступ, позволяя оценить и сравнить их производительность;

3 Вывод результатов: в конце тестов выводятся результаты времени выполнения операций;

4 Завершение работы: по завершении тестов программа корректно закрывает все ресурсы и сообщает о завершении работы.

4 ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.1 Запуск программы и процесс выполнения

Программа начинается с инициализации базы данных и создания файла database.dat. Запись и чтение данных проводится как с использованием асинхронных, так и синхронных методов. На рисунке 4.1 представлены результаты выполнения всех операций на консоли.

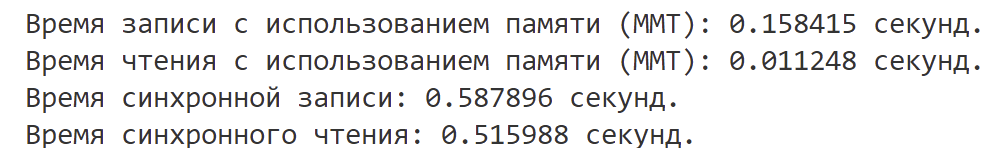
****

Рисунок 4.1 – Консоль с результатами выполнения потоков

Дополнительно, на рисунке 4.2 показано окно проводника Windows, где видно созданный файл database.dat и его размер, что подтверждает успешное выделение пространства на диске и фактическое создание файла.

****

Рисунок 4.2 – Сведения о файле database.dat в проводнике Windows

4.2 Описание работы и результатов

Для демонстрации различий в производительности между асинхронным и синхронным методами были проведены тесты с различными объемами данных: 10 000, 100 000 и 1 000 000 записей. Время выполнения операций чтения и записи фиксировалось для каждого объема данных и представлено в таблице 4.1.

Таблица 4.1 – Время выполнения операций для различных объемов данных

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Объем данных | Асинхронная запись (сек) | Синхронная запись (сек) | Асинхронное чтение (сек) | Синхронное чтение (сек) |
| 1000000 записей | 0.168827 | 0.596723 | 0.012421 | 0.497 |
| 10000000 записей | 1.66139 | 5.7988 | 0.133029 | 4.9768 |
| 100000000 записей | 33.1559 | 57.3492 | 8.38904 | 48.3424 |

Как видно из таблицы 4.1, время выполнения операций записи и чтения существенно различается между асинхронными и синхронными методами, особенно при увеличении объема данных. Эти различия обусловлены особенностями архитектуры асинхронного ввода-вывода, который позволяет системе выполнять несколько задач одновременно, не дожидаясь завершения каждой операции. Синхронный же метод требует полного завершения текущей операции перед началом следующей, что значительно замедляет выполнение при больших объемах данных.

Асинхронная запись показывает заметное преимущество по сравнению с синхронной. Например, для объема данных в 100 миллионов записей асинхронная запись занимает 33.1559 секунд, тогда как синхронная запись занимает 57.3492 секунд. Это демонстрирует значительное снижение времени выполнения, потому что асинхронный метод позволяет использовать ресурсы системы более эффективно. Система способна продолжать выполнение других операций, пока данные записываются в файл, что особенно важно для приложений, работающих в условиях высокой нагрузки и требующих минимальных задержек.

Асинхронное чтение также демонстрирует значительное улучшение производительности по сравнению с синхронным методом. Для 100 миллионов записей асинхронное чтение заняло 8.38904 секунд, в то время как синхронное чтение – 48.7558 секунд. Это подтверждает, что асинхронный метод способен минимизировать время ожидания операций ввода-вывода, что критически важно при обработке больших объемов информации. Разница в скорости выполнения операций становится еще более очевидной при увеличении объема данных, что делает асинхронный метод предпочтительным выбором для задач, требующих высокой производительности и быстрого отклика системы.

Эти результаты наглядно демонстрируют, что асинхронные методы ввода-вывода обеспечивают значительное улучшение общей производительности системы, особенно при работе с большими объемами данных. Асинхронные подходы позволяют сократить время простоя и более эффективно использовать вычислительные ресурсы, что делает их идеальным выбором для приложений, требующих высокой скорости обработки информации, таких как базы данных, серверы или другие системы с интенсивным доступом к данным.

# ВЫВОД

В ходе выполнения лабораторной работы были исследованы возможности и особенности работы с файлами в операционной системе Windows с использованием как асинхронных, так и синхронных методов ввода-вывода. Реализация и анализ применения технологии отображения файлов в память и асинхронных операций позволили оценить преимущества таких подходов по сравнению с традиционным синхронным доступом к данным.

Проведенные тесты на различных объемах данных подтвердили, что асинхронный метод ввода-вывода существенно сокращает время выполнения операций записи и чтения, особенно при увеличении объема данных. Это объясняется тем, что асинхронный подход позволяет системе эффективно использовать ресурсы, минимизируя время ожидания и обеспечивая возможность выполнения других операций параллельно. В свою очередь, использование отображения файлов в память обеспечивает быстрый доступ к данным, что также положительно сказывается на производительности системы.

Результаты экспериментов наглядно показали, что асинхронные методы и отображение файлов в память являются предпочтительными решениями для приложений, работающих с большими объемами данных и требующих высокой производительности. Эти подходы позволяют сократить задержки, повысить скорость обработки информации и более рационально использовать ресурсы системы.

Таким образом, работа не только позволила углубить знания в области системного программирования, но и предоставила практический опыт в оптимизации процессов ввода-вывода, что критически важно для разработки высокопроизводительных и масштабируемых приложений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Microsoft Documentation: Build desktop Windows apps using the Win32 API [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/>. – Дата доступа: 29.09.2024.

[2] Разработка приложений с помощью WinAPI [Электронный ресурс]. – Режим доступа: <https://shorturl.at/BDJW8>. – Дата доступа: 29.09.2024.

[3] Multithreading Tutorial - CodeProject [Электронный ресурс]. – Режим доступа: <https://www.codeproject.com/Articles/14746/Multithreading-Tutorial>. – Дата доступа: 29.09.2024.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код программы

#include <bits/stdc++.h>

#include <windows.h>

#define ll long long

#define ld long double

using namespace std;

using namespace chrono;

const ll RECORD\_SIZE = 256;

struct Record

{

ll id;

char name[248];

};

class Database

{

public:

Database(const string &filename, ll recordCount) : filename\_(filename), recordCount\_(recordCount)

{

hFile\_ = CreateFileA(

filename\_.c\_str(),

GENERIC\_READ | GENERIC\_WRITE,

0,

NULL,

OPEN\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL,

NULL);

LARGE\_INTEGER fileSize;

fileSize.QuadPart = recordCount\_ \* RECORD\_SIZE;

SetFilePointerEx(hFile\_, fileSize, NULL, FILE\_BEGIN);

SetEndOfFile(hFile\_);

hMap\_ = CreateFileMappingA(

hFile\_,

NULL,

PAGE\_READWRITE,

0,

0,

NULL);

data\_ = static\_cast<char \*>(MapViewOfFile(

hMap\_,

FILE\_MAP\_ALL\_ACCESS,

0,

0,

0));

}

~Database()

{

if (data\_)

UnmapViewOfFile(data\_);

if (hMap\_)

CloseHandle(hMap\_);

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

CloseHandle(hFile\_);

}

void writeRecord(ll index, const Record &record)

{

char \*ptr = data\_ + index \* RECORD\_SIZE;

memcpy(ptr, &record, sizeof(Record));

}

Record readRecord(ll index) const

{

Record record;

const char \*ptr = data\_ + index \* RECORD\_SIZE;

memcpy(&record, ptr, sizeof(Record));

return record;

}

private:

string filename\_;

ll recordCount\_;

HANDLE hFile\_ = INVALID\_HANDLE\_VALUE;

HANDLE hMap\_ = NULL;

char \*data\_ = NULL;

};

class SynchronousDatabase

{

public:

SynchronousDatabase(const string &filename, ll recordCount) : filename\_(filename), recordCount\_(recordCount)

{

hFile\_ = CreateFileA(

filename\_.c\_str(),

GENERIC\_READ | GENERIC\_WRITE,

0,

NULL,

OPEN\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL,

NULL);

LARGE\_INTEGER fileSize;

fileSize.QuadPart = recordCount\_ \* RECORD\_SIZE;

SetFilePointerEx(hFile\_, fileSize, NULL, FILE\_BEGIN);

SetEndOfFile(hFile\_);

}

~SynchronousDatabase()

{

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

CloseHandle(hFile\_);

}

void writeRecord(ll index, const Record &record)

{

LARGE\_INTEGER offset;

offset.QuadPart = index \* RECORD\_SIZE;

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

WriteFile(hFile\_, &record, sizeof(Record), NULL, NULL);

}

Record readRecord(ll index) const

{

Record record;

LARGE\_INTEGER offset;

offset.QuadPart = index \* RECORD\_SIZE;

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

ReadFile(hFile\_, &record, sizeof(Record), NULL, NULL);

return record;

}

private:

string filename\_;

ll recordCount\_;

HANDLE hFile\_ = INVALID\_HANDLE\_VALUE;

};

Record createRecord(ll id, const string &name)

{

Record record;

record.id = id;

memset(record.name, 0, sizeof(record.name));

strncpy\_s(record.name, name.c\_str(), sizeof(record.name) - 1);

return record;

}

int main()

{

system("cls");

const string filename = "database.dat";

const ll recordCount = 1000000;

Database db(filename, recordCount);

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

for (ll i = 0; i < recordCount; ++i)

{

db.writeRecord(i, createRecord(static\_cast<ll>(i), "Name\_" + to\_string(i)));

}

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

duration<double> duration = end - start;

cout << "Время записи с использованием памяти (MMT): " << duration.count() << " секунд." << endl;

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

for (ll i = 0; i < recordCount; ++i)

{

Record r = db.readRecord(i);

}

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

duration = end - start;

cout << "Время чтения с использованием памяти (MMT): " << duration.count() << " секунд." << endl;

SynchronousDatabase dbsync(filename, recordCount);

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

for (ll i = 0; i < recordCount; ++i)

{

dbsync.writeRecord(i, createRecord(static\_cast<ll>(i), "Name\_" + to\_string(i)));

}

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

duration = end - start;

cout << "Время синхронной записи: " << duration.count() << " секунд." << endl;

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

for (ll i = 0; i < recordCount; ++i)

{

Record r = dbsync.readRecord(i);

}

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

duration = end - start;

cout << "Время синхронного чтения: " << duration.count() << " секунд." << endl;

return 0;

}