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
Тимошевич К.С.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Реализация обработки содержимого файла данных путем отображения его в память и обращения к этой памяти. Варьирование количества потоков, получающих доступ к отображению (специальных мер для предотвращения коллизий не предусматривается), влияние на общею производительность.

Оценка эффективности (производительности) по сравнению с традиционным подходом (чтение обработка выгрузка), в том числе в многопоточном варианте.

Вариант обработки файла в работе: подсчет количества каждой буквы английского алфавита в текстовом файле.

2 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

В этом разделе будут рассмотрены основные функции программы, которые были задействованы для выполнения работы.

2.1 Традиционная обработка файла

Модуль traditional_file_processing реализует последовательное чтение файла и подсчет количества вхождений английских букв. Для работы используется функция CreateFile, открывающая файл, и ReadFile [1], которая считывает данные блоками фиксированного размера (1 МБ) в буфер. Затем данные из буфера анализируются функцией process_buffer, которая подсчитывает частоту встречаемости символов от 'a' до 'z' и 'A' до 'Z'. После завершения работы программа выводит статистику и общее время выполнения операции. Программный код представлен в приложении А [1].

2.2 Обработка файла с отображением в память

Модуль memory_mapped_file_processing использует технику отображения файла в память (memory-mapping), используя CreateFileMapping, MapViewOfFile, UnmapViewOfFile [2]. Это позволяет системе загрузить файл в адресное пространство процесса, что упрощает доступ к файлу и улучшает производительность за счет использования системных механизмов управления памятью. Данные из файла разделяются между потоками, и каждый поток анализирует свою часть файла, подсчитывая количество вхождений символов с помощью функции process_buffer.

2.3 Многопоточная традиционная обработка файла

Модуль multithreaded_traditional_file_processing реализует многопоточную обработку файла с использованием традиционного подхода чтения данных. Файл делится на несколько частей, и каждая часть файла читается и обрабатывается отдельным потоком. Потоки создаются с помощью CreateThread [3], а синхронизация выполняется с помощью WaitForMultipleObjects [3]. После завершения всех потоков результаты объединяются и выводятся на экран.

2.4 Замер времени выполнения

Для всех методов измерение времени выполнения осуществляется с помощью функции QueryPerformanceCounter [4], которая позволяет получить точное время начала и конца работы программы, что позволяет измерить время выполнения с высокой точностью.

3 ХОД ВЫПОЛНЕНИЯ ПРОГРАММЫ

3.1 Примеры выполнения задания

На рисунке 3.1 представлен результат выполнения функции traditional_file_processing (вывод количества каждой буквы английского алфавита в тексте и времени ее выполнения). На выходе программы отображается подробная статистика по каждому символу: начиная с буквы 'A' и заканчивая буквой 'Z', программа подсчитывает и выводит количество вхождений каждой буквы в файле. Кроме того, программа измеряет и выводит время выполнения этого подхода, что позволяет оценить его эффективность с точки зрения производительности.

```
Traditional approach - Statistics of occurrences of English letters:
A: 87829359
B: 14142777
C: 20494155
D: 44282151
E: 127451082
F: 18056550
G: 23438736
H: 66762861
I: 67760862
J: 931314
K: 9859647
L: 45588153
M: 31999605
N: 59017767
O: 81695151
P: 19164141
Q: 559764
R: 65850174
S: 61042224
T: 90644325
U: 28294644
V: 6145218
W: 25107954
X: 1860057
Y: 27160608
Z: 932136
Running time of the traditional approach: 6.16 seconds
```

Рисунок 3.1 - Результат функции traditional_file_processing

На рисунке 3.2 представлен результат работы функции memory_mapped_file_processing, использующей отображение файла в память и запуск на 10 потоках. Программа, как и в предыдущих случаях, подсчитывает количество вхождений каждой буквы английского алфавита от

'A' до 'Z' в содержимом файла. Однако, благодаря использованию отображения в память и многопоточности, обработка данных выполняется более эффективно. На выходе отображается подробная статистика по буквам, а также время выполнения этого метода, что позволяет оценить его производительность по сравнению с другими подходами.

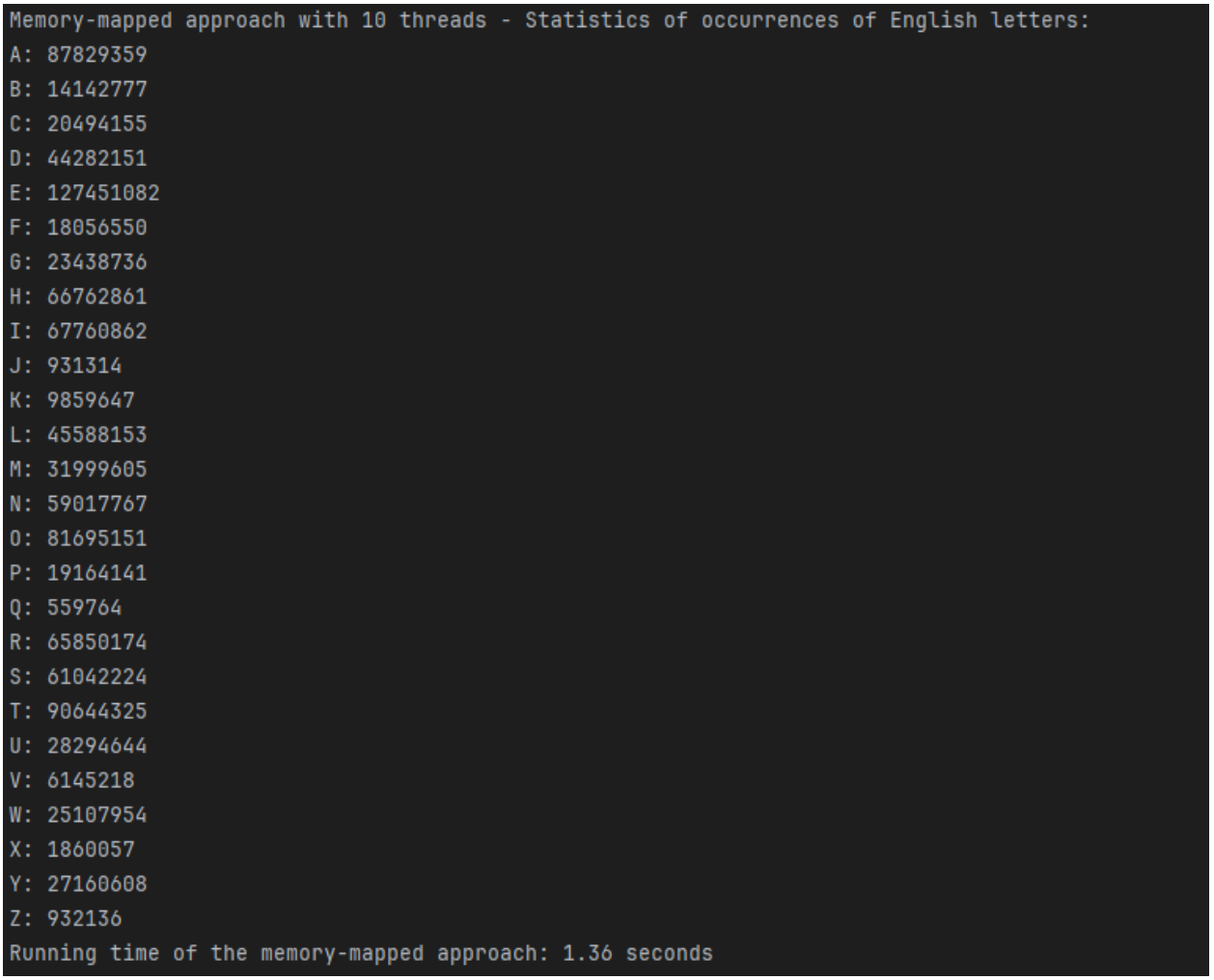

Рисунок 3.2 - Результат функции multithreaded_traditional_file_processing

На рисунке 3.3 представлен результат выполнения функции multithreaded_traditional_file_processing, которая реализует многопоточный подход к обработке файла. Как и в случае с традиционным методом, программа выводит статистику по каждому символу английского алфавита от 'A' до 'Z', подсчитывая количество вхождений каждой буквы в файле. Однако, в отличие от последовательного подхода, многопоточная реализация распределяет обработку данных между несколькими потоками, что позволяет значительно сократить время выполнения программы. На выходе также отображается время выполнения данного метода, что дает возможность сравнить его эффективность по сравнению с традиционным подходом.

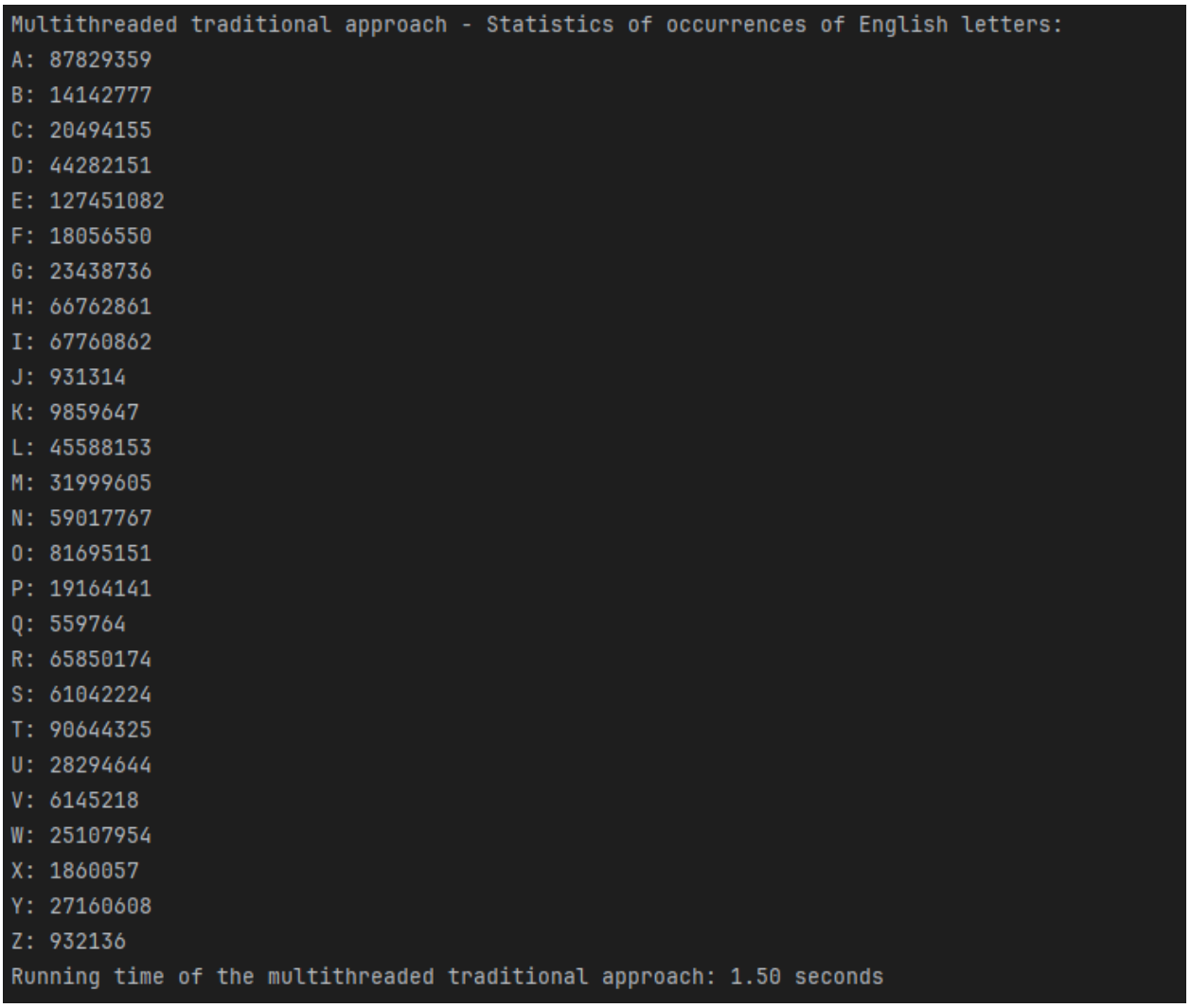

Рисунок 3.3 - Результат функции multithreaded_traditional_file_processing

ВЫВОД

В ходе выполнения лабораторной работы были изучены и реализованы различные подходы к обработке файлов на языке программирования C++ с использованием API Windows. Были рассмотрены как традиционный последовательный метод чтения и обработки данных, так метод отображения файла в память и многопоточная реализация. Тестирование и анализ результатов показали, что традиционный подход, несмотря на свою простоту и надежность, имеет ограниченную производительность из-за однопоточности. Он оказался неэффективным при работе с большими файлами, так как не использует возможности параллельного выполнения, что приводит к увеличению времени обработки.

Многопоточная реализация традиционного метода значительно сократила время выполнения программы за счет распределения работы между потоками. Метод отображения файла в память продемонстрировал наилучшую производительность, особенно при работе с большими файлами, так как позволяет напрямую работать с содержимым файла, избегая дополнительных операций чтения и выгрузки данных. При многопоточном доступе этот подход также показал высокую эффективность благодаря более быстрому доступу к данным в памяти.

Таким образом, сравнительный анализ методов показал, что для небольших файлов традиционный метод обработки может быть достаточным, однако для больших файлов многопоточные подходы и использование отображения в память оказываются более производительными. Выбор подхода зависит от характеристик задачи, таких как объем данных и требования к скорости обработки, что подтверждается проведенными экспериментами.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] API Win32 documentation [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/memory/memory-management-functions. – Дата доступа: 01.10.2024.

[2] WinAPI: Работа с файлами [Электронный ресурс]. – Режим доступа: http://zetblog.ru/winapi-rabota-s-faylami-osnovnye-funktsii.html – Дата доступа: 02.10.2024.

[3] API Win32 Threads [Электронный ресурс]. – Режим доступа: https://stackoverflow.com/questions/796217/what-is-the-difference-between-a-thread-and-a-fiber – Дата доступа: 02.10.2024.

[4] API Win32 Profileapi.h [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter. – Дата доступа: 13.09.2024.

ПРИЛОЖЕНИЕ А
(справочное)
Исходный код

```
#include <windows.h>
#include <stdio.h>
#include <string.h>

#define BUFFER_SIZE 1048576
#define NUM_THREADS 10

typedef struct {
    char *buffer;
    DWORD bytesRead;
    int letter_count[26];
} ThreadData;

double get_time_in_seconds() {
    LARGE_INTEGER frequency, currentTime;
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&currentTime);
    return (double)currentTime.QuadPart / frequency.QuadPart;
}

void process_buffer(const char *buffer, DWORD bytesRead, int letter_count[26])
{
    for (DWORD i = 0; i < bytesRead; i++) {
        char ch = buffer[i];
        if (ch >= 'A' && ch <= 'Z') {
            letter_count[ch - 'A']++;
        } else if (ch >= 'a' && ch <= 'z') {
            letter_count[ch - 'a']++;
        }
    }
}

DWORD WINAPI thread_function(LPVOID param) {
    ThreadData *data = (ThreadData *)param;
    process_buffer(data->buffer, data->bytesRead, data->letter_count);
    return 0;
}

void memory_mapped_file_processing(const char *filename) {
    HANDLE hFile = CreateFile(
            filename,
            GENERIC_READ,
            FILE_SHARE_READ,
            NULL,
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Error opening file: %d\n", GetLastError());
        return;
    }

    HANDLE hMapping = CreateFileMapping( hFile,NULL,PAGE_READONLY,0,0,NULL);

    if (hMapping == NULL) {
```

```
        printf("Error creating file mapping: %d\n", GetLastError());
        CloseHandle(hFile);
        return;
    }

    char *mappedView = (char *)MapViewOfFile(hMapping,FILE_MAP_READ,0,0,0);

    if (mappedView == NULL) {
        printf("Error mapping view of file: %d\n", GetLastError());
        CloseHandle(hMapping);
        CloseHandle(hFile);
        return;
    }

    DWORD fileSize = GetFileSize(hFile, NULL);
    int totalLetterCount[26] = { 0 };
    DWORD bytesPerThread = fileSize / NUM_THREADS;
    HANDLE threads[NUM_THREADS];
    ThreadData threadData[NUM_THREADS];

    double startTime = get_time_in_seconds();

    for (int i = 0; i < NUM_THREADS; i++) {
        threadData[i].buffer = &mappedView[i * bytesPerThread];
        threadData[i].bytesRead = (i == NUM_THREADS - 1) ? (fileSize - (i *
bytesPerThread)) : bytesPerThread;
        ZeroMemory(threadData[i].letter_count,
sizeof(threadData[i].letter_count));
        threads[i] = CreateThread(NULL, 0, thread_function, &threadData[i], 0,
NULL);
    }

    WaitForMultipleObjects(NUM_THREADS, threads, TRUE, INFINITE);

    for (int i = 0; i < NUM_THREADS; i++) {
        for (int j = 0; j < 26; j++) {
            totalLetterCount[j] += threadData[i].letter_count[j];
        }
    }
    double endTime = get_time_in_seconds();
    UnmapViewOfFile(mappedView);
    CloseHandle(hMapping);
    CloseHandle(hFile);

    printf("\nMemory-mapped approach with %d threads - Statistics of
occurrences of English letters:\n", NUM_THREADS);
    for (int i = 0; i < 26; i++) {
        printf("%c: %d\n", 'A' + i, totalLetterCount[i]);
    }
    printf("Running time of the memory-mapped approach: %.2f seconds\n",
endTime - startTime);
}

void traditional_file_processing(const char *filename) {
    HANDLE hFile;
    DWORD bytesRead = 0;
    int letter_count[26] = { 0 };
    double startTime, endTime;

    hFile =
CreateFile(filename,GENERIC_READ,0,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NUL
L);
```

```
    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Error opening file: %d\n", GetLastError());
        return;
    }
    startTime = get_time_in_seconds();
    char buffer[BUFFER_SIZE];
    while (TRUE) {
        BOOL result = ReadFile(hFile, buffer, sizeof(buffer), &bytesRead,
NULL);
        if (!result || bytesRead == 0) {
            break;
        }
        process_buffer(buffer, bytesRead, letter_count);
    }

    endTime = get_time_in_seconds();
    CloseHandle(hFile);

    printf("\nTraditional approach - Statistics of occurrences of English
letters:\n");
    for (int i = 0; i < 26; i++) {
        printf("%c: %d\n", 'A' + i, letter_count[i]);
    }
    printf("Running time of the traditional approach: %.2f seconds\n", endTime
- startTime);
}

void multithreaded_traditional_file_processing(const char *filename) {
    HANDLE hFile;
    DWORD bytesRead = 0;
    int totalLetterCount[26] = { 0 };
    double startTime, endTime;

    hFile =
CreateFile(filename,GENERIC_READ,0,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NUL
L);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Error opening file: %d\n", GetLastError());
        return;
    }

    DWORD fileSize = GetFileSize(hFile, NULL);
    DWORD bytesPerThread = fileSize / NUM_THREADS;
    HANDLE threads[NUM_THREADS];
    ThreadData threadData[NUM_THREADS];

    startTime = get_time_in_seconds();

    for (int i = 0; i < NUM_THREADS; i++) {
        char *buffer = new char[bytesPerThread];
        ReadFile(hFile, buffer, bytesPerThread, &bytesRead, NULL);
        threadData[i].buffer = buffer;
        threadData[i].bytesRead = bytesRead;
        ZeroMemory(threadData[i].letter_count,
sizeof(threadData[i].letter_count));
        threads[i] = CreateThread(NULL, 0, thread_function, &threadData[i], 0,
NULL);
    }

    WaitForMultipleObjects(NUM_THREADS, threads, TRUE, INFINITE);

    for (int i = 0; i < NUM_THREADS; i++) {
```

```
        for (int j = 0; j < 26; j++) {
            totalLetterCount[j] += threadData[i].letter_count[j];
        }
        delete[] threadData[i].buffer;
    }

    endTime = get_time_in_seconds();
    CloseHandle(hFile);

    printf("\nMultithreaded traditional approach - Statistics of occurrences of
English letters:\n");
    for (int i = 0; i < 26; i++) {
        printf("%c: %d\n", 'A' + i, totalLetterCount[i]);
    }
    printf("Running time of the multithreaded traditional approach: %.2f
seconds\n", endTime - startTime);
}

int main() {
    const char *filename = "D:\\TEST\\text.txt";
    traditional_file_processing(filename);
    memory_mapped_file_processing(filename);
    multithreaded_traditional_file_processing(filename);
    return 0;
}
```