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 №1
на тему

РАБОТА С ФАЙЛАМИ

Студент Дмитрук Б.Я.

Преподаватель Гриценко Н.Ю.

Минск 2024

СОДЕРЖАНИЕ

1 ФОРМУЛИРОВКА ЗАДАЧИ

Целью выполнения лабораторной работы являются Возобновление, закрепление и развитие навыков программирования приложений для операционной системы Windows. Изучение концепций вычислительных процессов и потоков, а также их реализации в среде Windows. Рассмотрение основных этапов жизненного цикла процессов и потоков, включая их создание, завершение, получение и изменение состояния. Анализ типичного использования многозадачности и многопоточности в приложениях.

В качестве задачи необходимо реализовать программу, позволяющую пользователю выбирать файл для чтения и количество потоков (в том числе возможность использования одного потока). В ходе работы будет проведена оценка времени выполнения чтения и анализ зависимости времени выполнения от размера файла и количества потоков, а также зависимость от синхронной и асинхронной реализации.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Потоки — это базовый элемент выполнения внутри процесса, который позволяет программе выполнять несколько задач одновременно. Они представляют собой независимые пути выполнения, разделяющие ресурсы и адресное пространство процесса, но при этом работающие параллельно. Потоки в операционной системе Windows позволяют программам распределять задачи и улучшать производительность за счет многозадачности. Каждому потоку назначается своя задача, что позволяет, например, одному потоку обрабатывать пользовательский интерфейс, а другому — выполнять сложные вычисления. Благодаря этому достигается высокая скорость отклика приложения и оптимизируется использование ресурсов системы.

Потоки взаимодействуют через общую память и требуют синхронизации, так как доступ к одним и тем же ресурсам может привести к конфликтам. Для этого в Windows предоставляются различные механизмы, такие как мьютексы, критические секции и события. Кроме того, в программировании на Windows часто различают два типа ввода-вывода: синхронный и асинхронный. В синхронном вводе-выводе поток ждет завершения операции, прежде чем продолжить выполнение, тогда как асинхронный ввод-вывод позволяет потоку продолжить выполнение других задач, пока система выполняет операцию. Выбор между этими типами ввода-вывода зависит от специфики задачи и требований к производительности приложения, что делает концепцию потоков гибкой и подходящей для множества сценариев использования.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Для получения статистических данных создается csv документ, в котором в виде таблицы из двух столбцов записывается время потраченное на чтение файла и количество потоков, использующихся для чтения.

Понятие «синхронности» и «асинхронности» рассматривались в рамках данной лабораторной работы как синхронный и асинхронный доступ к читаемому файлу. Осуществляется синхронный и асинхронный режим через корректировку аргумента метода *CreateFileA*, который отвечает за режим доступа к файлу. Значение которого *FILE_SHARE_READ* разрешало асинхронное чтение файла, а 0 запрещало соответственно.

3.1 Синхронное чтение файла

Результаты выполнения программы в синхронном режиме можно увидеть на четырех графиках: рисунок 3.1, рисунок 3.2, рисунок 3.3, рисунок 3.4 – отражающих показатели измерений при размерах файла 1 543 417, 7 712 080, 15 424 162 и 61 696 648 байт соответственно.

Рисунок 3.1 – Зависимость времени выполнения от числа задействованных потоков при размере файла 1 543 417 байт.

Рисунок 3.2 – Зависимость времени выполнения от числа задействованных потоков при размере файла 7 712 080 байт.

Рисунок 3.3 – Зависимость времени выполнения от числа задействованных потоков при размере файла 15 424 162 байт.

Рисунок 3.4 – Зависимость времени выполнения от числа задействованных потоков при размере файла 61 696 648 байт.

3.2 Асинхронное чтение файла

Результаты выполнения программы в синхронном режиме можно увидеть на четырех графиках: рисунок 3.5, рисунок 3.6, рисунок 3.7, рисунок 3.8 – отражающих показатели измерений при размерах файла 1 543 417, 7 712 080, 15 424 162 и 61 696 648 байт соответственно.

Рисунок 3.5 – Зависимость времени выполнения от числа задействованных потоков при размере файла 1 543 417 байт.

Рисунок 3.6 – Зависимость времени выполнения от числа задействованных потоков при размере файла 7 712 080 байт.

Рисунок 3.7 – Зависимость времени выполнения от числа задействованных потоков при размере файла 15 424 162 байт.

Рисунок 3.8 – Зависимость времени выполнения от числа задействованных потоков при размере файла 61 696 648 байт.

ЗАКЛЮЧЕНИЕ

В ходе данной работы был получен программный продукт, производящий чтение файла несколькими потоками, а также получена зависимость времени чтения файла от количества потоков, размера файла, синхронного либо асинхронного режима работы.

По результатам графиков, приведенных в пунктах 3.1 и 3.2 данной работы, можно заключить, что оптимальное время чтения файла достигается при использовании от 5 до 10 потоков вне зависимости от размеров файла. При этом при синхронном режиме чтения наблюдается более стремительный рост времени, затрачиваемого на чтение чем при асинхронном.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Доступ к данным и хранилище Windows – Электронный ресурс. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/_fs/

[2] Процессы и потоки Windows – Электронный ресурс. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/_processthreadsapi/

ПРИЛОЖЕНИЕ А
(обязательное)
Исходный код программы

```
#include <windows.h>
#include <iostream>
#include <vector>
#include <string>
#include <fstream>

struct ThreadParams {
    HANDLE hFile;
    DWORD startOffset;
    DWORD bytesToRead;
    std::vector<char>* buffer;
    int threadId;
    OVERLAPPED overlapped;
};

std::vector<int> threads;
std::vector<double> times;

void CALLBACK ReadComplete(DWORD dwErrorCode, DWORD
dwNumberOfBytesTransferred, LPOVERLAPPED lpOverlapped) {
    ThreadParams* params = reinterpret_cast<ThreadParams*>(lpOverlapped-
>hEvent);
    delete params;
}

DWORD WINAPI readFileChunk(LPVOID lpParams) {
    ThreadParams* params = static_cast<ThreadParams*>(lpParams);
    params->overlapped.hEvent = reinterpret_cast<HANDLE>(params);
    params->overlapped.Offset = params->startOffset;
    params->overlapped.OffsetHigh = 0;
    params->buffer->resize(params->bytesToRead);

    BOOL result = ReadFileEx(params->hFile, params->buffer->data(), params-
>bytesToRead, &params->overlapped, ReadComplete);
    if (!result) {
        DWORD error = GetLastError();
        std::cerr << "Error initiating asynchronous read for thread " <<
params->threadId << ". Error code: " << error << "\n";
        delete params;
        return 1;
    }
    return 0;
}

bool fileExists(const std::string& fileName) {
    std::wstring wideFileName(fileName.begin(), fileName.end());
    DWORD fileAttributes = GetFileAttributes(wideFileName.c_str());
    return fileAttributes != INVALID_FILE_ATTRIBUTES;
}

void read_file_multithread(std::string fileName, int numThreads) {
    HANDLE hFile = CreateFileA(fileName.c_str(), GENERIC_READ,
/*FILE_SHARE_READ*/ 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        std::cerr << "Error opening file.\n";
        return;
```

```
    }
    else
    {
        std::cout << "\nfile descriptor received\n";
    }

    LARGE_INTEGER fileSize;
    GetFileSizeEx(hFile, &fileSize);

    DWORD chunkSize = static_cast<DWORD>(fileSize.QuadPart / numThreads);
    DWORD lastChunkSize = chunkSize + static_cast<DWORD>(fileSize.QuadPart %
numThreads);

    std::vector<HANDLE> threadHandles(numThreads);
    std::vector<std::vector<char>> buffers(numThreads);

    LARGE_INTEGER frequency, start, end;
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&start);

    for (int i = 0; i < numThreads; ++i) {
        DWORD currentChunkSize = (i == numThreads - 1) ? lastChunkSize :
chunkSize;
        ThreadParams* params = new ThreadParams{ hFile, i * chunkSize,
currentChunkSize, &buffers[i], i + 1, {} };

        threadHandles[i] = CreateThread(NULL, 0, readFileChunk, params, 0,
NULL);
        if (threadHandles[i] == NULL) {
            std::cerr << "Error while creating thread.\n";
            delete params;
            CloseHandle(hFile);
            return;
        }
    }

    WaitForMultipleObjects(numThreads, threadHandles.data(), TRUE, INFINITE);

    for (HANDLE threadHandle : threadHandles) {
        CloseHandle(threadHandle);
    }

    CloseHandle(hFile);

    QueryPerformanceCounter(&end);
    double elapsedTime = static_cast<double>(end.QuadPart - start.QuadPart) /
frequency.QuadPart;

    std::vector<char> finalData;
    for (const auto& buffer : buffers) {
        finalData.insert(finalData.end(), buffer.begin(), buffer.end());
    }

    //std::cout << "File reading time: " << elapsedTime << " seconds\n";
    //std::cout << "Reading file size: " << finalData.size() << " bytes\n";
    times.push_back(elapsedTime);
    threads.push_back(numThreads);
}

void writeDataToCSV(const std::string& filename, const std::vector<int>&
threadCounts, const std::vector<double>& times) {
    std::ofstream outFile(filename);
```

```
    if (!outFile.is_open()) {
        std::cerr << "File open error\n";
        return;
    }

    outFile << "Количество потоков,Время\n";

    for (size_t i = 0; i < threadCounts.size(); ++i) {
        outFile << threadCounts[i] << "," << times[i] << "\n";
    }

    outFile.close();
    std::cout << "Data successfully writed" << filename << "\n";
}

int main() {
    std::string fileName = "file4.txt";
    int numThreads;


    for (numThreads = 1; numThreads < 60; ++numThreads) {
        //std::cout << "Enter file name to read: ";
        //std::cin >> fileName;
        //std::cout << "Enter thread count: ";
        //std::cin >> numThreads;
        if (!fileExists(fileName)) {
            std::cout << "File does not exist. Please enter a valid file
name.\n";
            continue;
        }

        read_file_multithread(fileName, numThreads);
        writeDataToCSV("result.csv", threads, times);
    }

    return 0;
}
```