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

**УПРАВЛЕНИЕ ПРОЦЕССАМИ, ПОТОКАМИ, НИТЯМИ**

Выполнил: Дмитрук Б.Я.

Проверил: Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка Задачи 3](#_Toc179481879)

[2 Краткие теоритические сведения 4](#_Toc179481880)

[3 Описание функций программы 5](#_Toc179481881)

[3.1 Синхронное чтение файла 5](#_Toc179481882)

[3.2 Асинхронное чтение файла 7](#_Toc179481883)

[Заключение 10](#_Toc179481884)

[Список использованных источников 11](#_Toc179481885)

[Приложение А (обязательное) 12](#_Toc179481886)

# **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. В качестве справочных материалов были использованы статьи документации *Windows* под названиями «Доступ к данным и хранилище Windows» [1] и «Процессы и потоки Windows» [2].

## **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 разработан программный продукт, который выполняет многопоточное чтение файлов и позволяет получить зависимость времени чтения от ряда параметров, таких как количество потоков, размер файла и режим работы (синхронный или асинхронный). Этот программный продукт стал инструментом для изучения влияния различных факторов на производительность многопоточного чтения файлов.

Анализ результатов, представленных в пунктах 3.1 и 3.2, показал, что оптимальное время чтения файла достигается при использовании от 5 до 10 потоков, независимо от размера файла. В этом диапазоне количество потоков обеспечивает баланс между параллельной загрузкой и затратами на управление потоками. Превышение указанного количества потоков не даёт заметных преимуществ, поскольку время чтения остаётся примерно постоянным или даже увеличивается из-за накладных расходов, связанных с синхронизацией.

Кроме того, сравнительный анализ синхронного и асинхронного режимов чтения показал, что при синхронном подходе наблюдается более быстрое возрастание времени выполнения с увеличением объема данных. Асинхронный режим работы оказался более эффективным, так как позволяет потокам не ждать завершения операций чтения, что минимизирует время ожидания и улучшает общую производительность. Эти результаты демонстрируют важность выбора подходящего режима для достижения максимальной производительности в зависимости от конкретных требований и условий задачи.

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;

}