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

БГУИР 1-40 04 01

Выполнил: студент гр. 253503
Тимошевич К.С.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Целью лабораторной работы является освоение навыков многопоточной работы с файлами. Была поставлена следующая задача: прочитать файл несколькими потоками и осуществить «сборку» результата. Количество потоков и файл для чтения - выбор пользователя. Оценить время и зависимость от начальных параметров.

2 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

2.1 Инициализация счетчика загрузки процессора

Этот модуль отвечает за настройку счетчика производительности, который будет отслеживать общую загрузку процессора. Это достигается с помощью функций PdhOpenQuery, PdhAddEnglishCounter и PdhCollectQueryData из библиотеки Performance Data Helper (PDH).

2.2 Основная работа программы

Этот модуль начинается с запроса у пользователя пути к файлу, который будет открыт для чтения. Затем программа запрошенное количество потоков для чтения файла. Каждый поток читает свою часть файла. Это достигается с помощью функций CreateFile, GetFileSizeEx, CreateThread и WaitForMultipleObjects.

2.3 Замер нагрузки процессора

Во время работы основного модуля программы, программа также отслеживает текущую загрузку процессора. Это достигается с помощью функции getCurrentCpuLoad, которая использует функции PdhCollectQueryData и PdhGetFormattedCounterValue для получения текущей загрузки процессора.

2.4 Вывод результатов

После завершения всех потоков, программа выводит время чтения и текущую загрузку процессора. Затем программа закрывает файл и завершает свою работу. Это достигается с помощью функций QueryPerformanceCounter, CloseHandle и стандартных функций вывода C++.

3 Ход выполнения программы

На рисунке 3.1 представлен результат выполнения программы при выборе запуска программы на одном потоке (вывод загруженности CPU и времени выполнения).

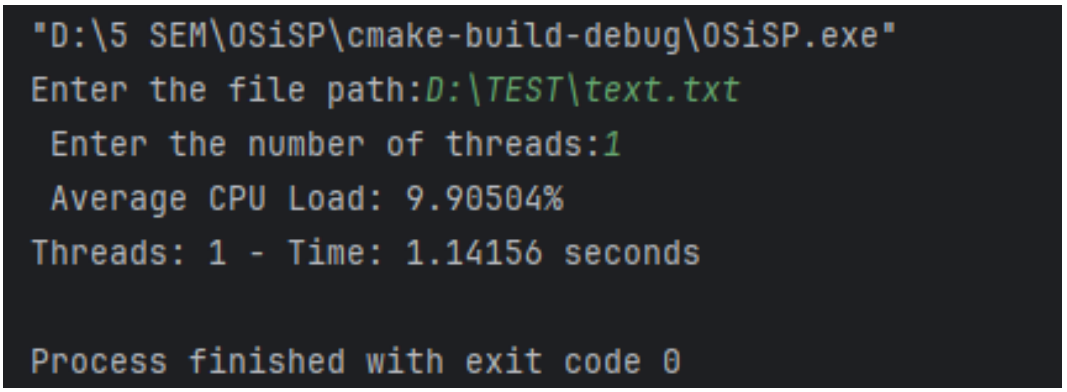

Рисунок 3.1 - Результат выполнения программы

Результат работы программы при выборе пользователем запустить на 16 потоках на рисунке 3.2.

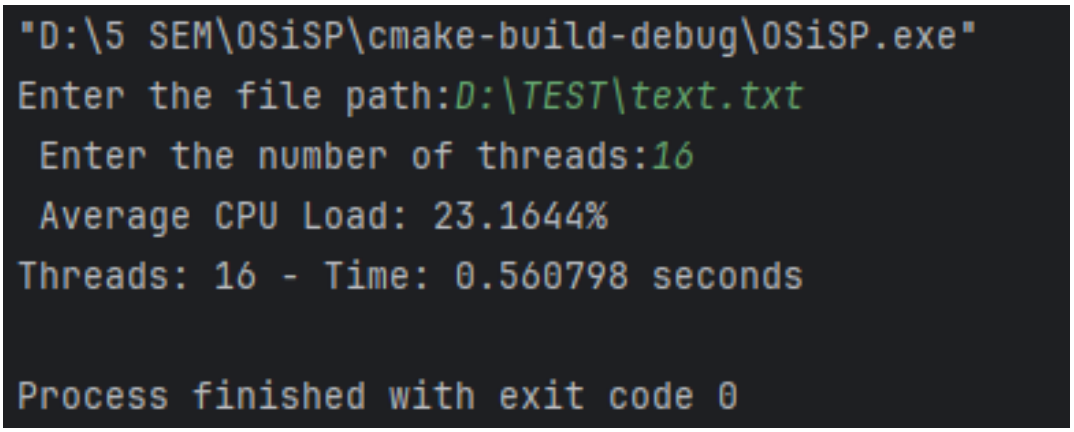

Рисунок 3.2 - Результат выполнения программы

В программе предусмотрена обработка ошибок, например, на рисунке 3.3 изображена обработка некорректного выбора количества потоков, в результате чего выводится сообщение об ошибке и программа завершается.

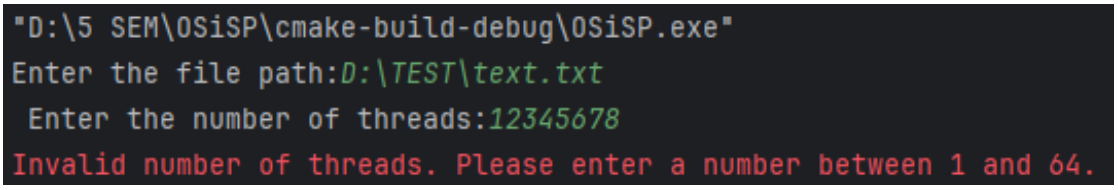

Рисунок 3.3 - Обработка некорректного количества потоков

Обработка ошибки при выборе файла представлена на рисунке 3.4

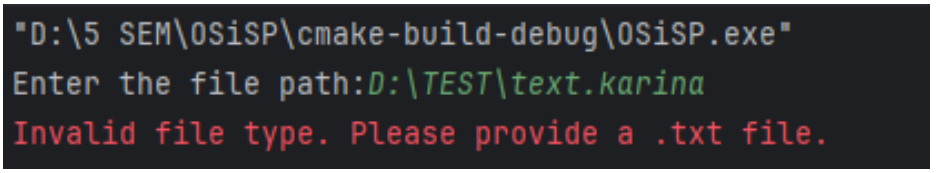

Рисунок 3.4 - Обработка ошибки выбора файла

На рисунке 3.5 изображена обработка ошибки в случае, если пользователь ввел количество потоков большее, чем количество байт для чтения.

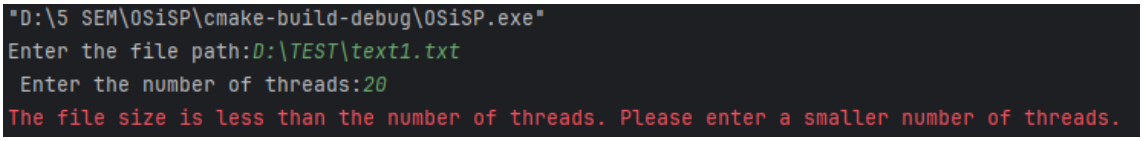

Рисунок 3.5 - Обработка ошибки выбора потоков

ВЫВОД

В ходе выполнения лабораторной работы были освоены навыки многопоточной работы с файлами. Была реализована программа, которая читает файл с использованием нескольких потоков и осуществляет «сборку» результата.

В процессе работы было обнаружено, что многопоточное чтение может быть эффективным при работе с большими файлами, поскольку оно позволяет распределить нагрузку на процессор и ускорить процесс чтения. Однако при работе с небольшими файлами использование многопоточности может не давать значительного преимущества, поскольку накладные расходы на создание и управление потоками могут превышать выигрыш от параллельного выполнения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] API Win32 documentation [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/ – Дата доступа: 13.09.2024

[2] WinAPI: Работа с файлами [Электронный ресурс]. – Режим доступа: http://zetblog.ru/winapi-rabota-s-faylami-osnovnye-funktsii.html – Дата доступа: 14.09.2024

ПРИЛОЖЕНИЕ А
(справочное)
Исходный код

```
#include <windows.h>

#include <pdh.h>

#include <vector>

#include <iostream>

#include <filesystem>

#include <numeric>

#include <mutex>


std::vector<double> cpuLoads;

std::mutex cpuLoadsMutex;


//Performance Data Helper, дескриптор запроса производительности и счетчика процессора

PDH_HQUERY cpuQuery;

PDH_HCOUNTER cpuTotal;


void initCpuLoadCounter() {

    PdhOpenQuery(NULL, NULL, &cpuQuery);

    PdhAddEnglishCounter(cpuQuery, "\\Processor(_Total)\\% Processor Time", NULL, &cpuTotal);

    PdhCollectQueryData(cpuQuery);

}


double getCurrentCpuLoad() {

    PDH_FMT_COUNTERVALUE counterVal;

    PdhCollectQueryData(cpuQuery);

    PdhGetFormattedCounterValue(cpuTotal, PDH_FMT_DOUBLE, NULL, &counterVal);

    return counterVal.doubleValue;

}


struct ThreadParams {
```

```
    HANDLE hFile;
    LARGE_INTEGER start;
    LARGE_INTEGER end;
    std::vector<char>* result;
};

DWORD WINAPI ThreadFunc(LPVOID lpParam) {
    auto params = static_cast<ThreadParams*>(lpParam);
    DWORD bytesRead;
    DWORD toRead = params->end.QuadPart - params->start.QuadPart;
    params->result->resize(toRead);
    SetFilePointerEx(params->hFile, params->start, NULL, FILE_BEGIN);

    if (!ReadFile(params->hFile, &(*params->result)[0], toRead, &bytesRead,
NULL)) {
        std::cerr << "Failed to read file.\n";
        return 1;
    }
    double cpuLoadAfter = getCurrentCpuLoad();

    {
        std::lock_guard<std::mutex> lock(cpuLoadsMutex);
        cpuLoads.push_back(cpuLoadAfter);
    }
    return 0;
}

int main() {
    initCpuLoadCounter();
    std::string filename;
    std::cout << "Enter the file path: ";
    std::cin >> filename;

    std::filesystem::path filePath(filename);
    if (filePath.extension() != ".txt") {
```

```
        std::cerr << "Invalid file type. Please provide a .txt file.\n";
        return 1;
    }

    HANDLE hFile = CreateFile(filename.c_str(), GENERIC_READ, FILE_SHARE_READ,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        std::cerr << "Unable to open file.\n";
        return 1;
    }

    LARGE_INTEGER fileSize;
    GetFileSizeEx(hFile, &fileSize);

    int threadCountInp;
    std::cout << "Enter the number of threads: ";
    std::cin >> threadCountInp;
    if (threadCountInp < 1 || threadCountInp > 64) {
        std::cerr << "Invalid number of threads. Please enter a number between
1 and 64.\n";
        return 1;
    }

    if (fileSize.QuadPart < threadCountInp) {
        std::cerr << "The file size is less than the number of threads. Please
enter a smaller number of threads.\n";
        return 1;
    }

    for (int threadCount = threadCountInp; threadCount <= threadCountInp;
++threadCount) {
        LARGE_INTEGER partSize;
        partSize.QuadPart = fileSize.QuadPart / threadCount;
        std::vector<HANDLE> threads(threadCount);
        LARGE_INTEGER startTime, endTime, freq;
        QueryPerformanceFrequency(&freq);
```

```
        QueryPerformanceCounter(&startTime);


        std::vector<ThreadParams*> paramsList(threadCount);


        for (int i = 0; i < threadCount; ++i) {
            ThreadParams* params = new ThreadParams;
            params->hFile = hFile;
            params->start.QuadPart = i * partSize.QuadPart;
            params->end.QuadPart = (i == threadCount - 1) ? fileSize.QuadPart :
params->start.QuadPart + partSize.QuadPart;
            params->result = new std::vector<char>(params->end.QuadPart -
params->start.QuadPart);


            threads[i] = CreateThread(NULL, 0, ThreadFunc, params, 0, NULL);
            paramsList[i] = params;
        }
        WaitForMultipleObjects(threadCount, &threads[0], TRUE, INFINITE);
        double averageCpuLoad = std::accumulate(cpuLoads.begin(),
cpuLoads.end(), 0.0) / cpuLoads.size();
        std::cout << "Average CPU Load: " << averageCpuLoad << "%\n";


        QueryPerformanceCounter(&endTime);
        double elapsedTime = static_cast<double>(endTime.QuadPart -
startTime.QuadPart) / freq.QuadPart;
        std::cout << "Threads: " << threadCount << " - Time: " << elapsedTime
<< " seconds\n";


        for (int i = 0; i < threadCount; ++i) {
            CloseHandle(threads[i]);
            delete paramsList[i]->result;
            delete paramsList[i];
        }
    }
    CloseHandle(hFile);
    return 0;
}
```