

Министерство науки и высшего образования Российской Федерации
Пензенский государственный университет
Кафедра «Вычислительная техника»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К курсовой работе
по курсу «Технологии программирования»
на тему «Многопоточная обработка Markdown-документов с
использованием DLL на C++»

Выполнил студент группы 23ВВИ1:

Пыльнов Н.Н.

Принял:

Сорокин А.А.

Патунин Д.В.

хорошо *Д.В. Патунин*
24.12.25

Пенза 2025 г.

Содержание

Введение.....	3
Постановка задачи.....	3
Теоретическая часть задания.....	5
Практическая часть.....	5
Тестирование и результаты.....	6
Заключение.....	8
Список литературы.....	9
Приложение А. Листинг программы.....	10

Введение

Современные приложения всё чаще сталкиваются с необходимостью обработки большого количества документов в реальном времени. Для повышения производительности таких систем используется многопоточность, которая позволяет выполнять задачи параллельно, экономя время и ресурсы.

Целью данной курсовой работы является разработка программы на C++ с использованием динамически подключаемой библиотеки (DLL), которая выполняет многопоточную обработку текстовых файлов формата Markdown (.md) с последующим извлечением структурированных данных (заголовков, списков, ссылок, параграфов) и записью результатов в общий выходной файл.

Задачи работы:

1. Изучение основ многопоточного программирования в C++.
2. Создание DLL-библиотеки для парсинга Markdown-документов.
3. Разработка многопоточной системы чтения и обработки файлов.
4. Реализация очереди задач для безопасного взаимодействия потоков.
5. Тестирование производительности и проверка корректности обработки.

Постановка задачи

Разработать приложение на C++20, которое:

- Сканирует указанную директорию и находит все файлы формата Markdown (.md).
- Загружает из DLL функции для парсинга Markdown-документов.
- Асинхронно (параллельно) читает найденные файлы.
- Выполняет для каждого файла парсинг: извлекает заголовки (#), списки (*, -, 1.), ссылки ([]()), параграфы.
- Собирает результаты парсинга и записывает их в единый выходной файл (result.txt) в структурированном формате.
- Работает на Windows без сторонних библиотек, совместимо с Visual Studio (C++20).

Требования к производительности: программа должна обрабатывать множество файлов быстрее, чем последовательная реализация, за счёт параллельной обработки (чтение + парсинг в рабочих потоках).

Теоретическая часть задания

Многопоточная обработка позволяет одновременно выполнять несколько задач, используя несколько потоков выполнения. В C++ для работы с потоками используются:

- **std::thread** — низкоуровневый поток выполнения.
- **std::async** / **std::future** — высокоуровневая абстракция для асинхронных операций.
- **std::mutex** / **std::lock_guard** / **std::unique_lock** — для синхронизации доступа к общим ресурсам.
- **std::condition_variable** — для оповещения потоков о новых данных.
- **std::atomic** — для атомарных операций.

DLL (Dynamic Link Library) — это библиотека, которая подключается к программе во время выполнения. Это позволяет разделить логику обработки данных и основную логику приложения, упрощая поддержку и расширение.

В данной работе реализована архитектура producer-consumer с использованием DLL:

Producer — поток, сканирующий папку и добавляющий пути к файлам в очередь.

Consumer — несколько рабочих потоков, которые загружают Markdown-парсер из DLL, обрабатывают файлы и записывают результат в общий файл.

- Такой подход позволяет:
- Параллельно обрабатывать несколько Markdown-файлов.
- Изолировать логику парсинга в отдельную библиотеку.
- Обеспечить безопасный доступ к общему файлу с помощью мьютексов.

Практическая часть

Архитектура программы

Программа состоит из следующих компонентов:

1. MainApp — основное приложение, управляющее потоками и очередью задач.
2. MarkdownParser.dll — библиотека, содержащая функции для парсинга Markdown-документов.
3. FileProcessor — класс для многопоточной обработки файлов.
4. ResultWriter — модуль для записи результатов в файл.

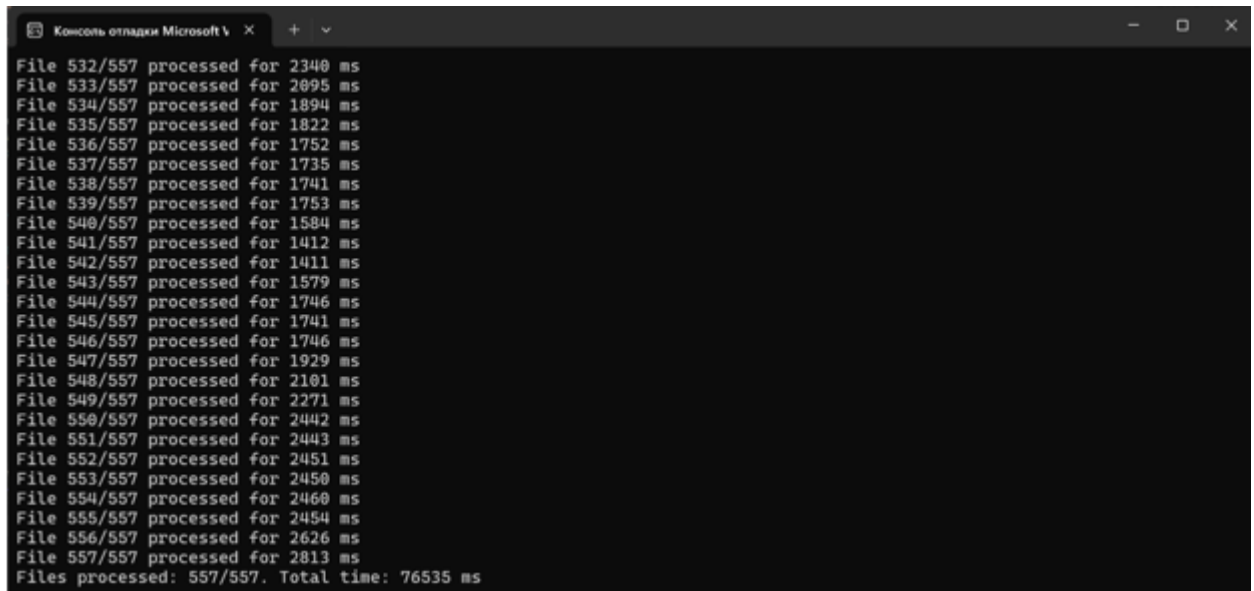
Рабочий алгоритм:

1. Приложение загружает функции из MarkdownParser.dll.
2. Поток-сканировщик находит все .md файлы в указанной папке и добавляет их в очередь.
3. Рабочие потоки берут файлы из очереди, вызывают функцию парсинга из DLL и сохраняют результат.
4. Все потоки синхронизируются через мьютексы и условные переменные.
5. Результаты записываются в result.txt.

Тестирование и результаты

Для проверки работоспособности программы было использовано 557 файлов формата Markdown разного размера.

Результат выполнения на многопоточном процессоре:



```
File 532/557 processed for 2340 ms
File 533/557 processed for 2095 ms
File 534/557 processed for 1894 ms
File 535/557 processed for 1822 ms
File 536/557 processed for 1752 ms
File 537/557 processed for 1735 ms
File 538/557 processed for 1741 ms
File 539/557 processed for 1753 ms
File 540/557 processed for 1584 ms
File 541/557 processed for 1412 ms
File 542/557 processed for 1411 ms
File 543/557 processed for 1579 ms
File 544/557 processed for 1746 ms
File 545/557 processed for 1741 ms
File 546/557 processed for 1746 ms
File 547/557 processed for 1929 ms
File 548/557 processed for 2101 ms
File 549/557 processed for 2271 ms
File 550/557 processed for 2442 ms
File 551/557 processed for 2443 ms
File 552/557 processed for 2451 ms
File 553/557 processed for 2450 ms
File 554/557 processed for 2460 ms
File 555/557 processed for 2454 ms
File 556/557 processed for 2626 ms
File 557/557 processed for 2813 ms
Files processed: 557/557. Total time: 76535 ms
```

Рисунок 1 – Результат работы программы

Заключение

В ходе работы была разработана программа на C++, реализующая многопоточную обработку Markdown-файлов с использованием DLL. Программа успешно:

- Читает файлы формата .md.
- Загружает парсер из динамической библиотеки.
- Извлекает структурированные элементы (заголовки, списки, ссылки, параграфы).
- Записывает результаты в общий файл без конфликтов доступа.

Использование DLL позволило разделить логику парсинга и основного приложения, а многопоточность обеспечила высокую производительность при обработке большого количества файлов.

Список литературы

1. Бьёрн Страуструп. Программирование на C++. М.: Питер, 2020.
2. Nicolai M. Josuttis. The C++ Standard Library: A Tutorial and Reference. Addison-Wesley, 2019.
3. Anthony Williams. C++ Concurrency in Action. Manning Publications, 2021.
4. Документация C++17: <https://en.cppreference.com/w/cpp/thread>
5. Markdown Guide: <https://www.markdownguide.org/>

Приложение А. Листинг программы.

main.cpp

```
#include "FileProcAsync.h"
#include "DLLLoader.h"
#include <iostream>
#include <fstream>

DLLLoader mdParser;
std::mutex outputMutex;

void processFileContent(const std::string& content, std::ofstream& outfile) {
    std::list<std::pair<std::string, std::string>> elements;

    // Используем DLL для парсинга
    mdParser.parse(content, elements);

    // Записываем результаты
    std::lock_guard<std::mutex> lock(outputMutex);
    for (const auto& elem : elements) {
        outfile << elem.first << ": " << elem.second << "\n";
    }
    outfile << "---\n";
}

int main() {
    // Загружаем DLL
    if (!mdParser.load("MarkdownParser.dll")) {
        std::cerr << "Failed to load MarkdownParser.dll\n";
        return 1;
    }

    std::string folder = "C:\\test\\md_files";
    std::ofstream outfile("result.txt");

    if (!outfile.is_open()) {
        std::cerr << "Cannot open output file\n";
        return 1;
    }
}
```

```

    }

    unsigned int threads = std::thread::hardware_concurrency();
    FileProcAsync processor(folder, threads, processFileContent, outfile);

    std::cout << "Starting processing...\n";
    processor.start();
    processor.wait();

    outfile.close();
    std::cout << "Done!\n";

    return 0;
}

```

Parser.cpp

```

#define MARKDOWN_EXPORTS
#include "Parser.h"
#include <regex>
#include <sstream>

MD_API void parse_markdown(const char* content,
                           std::list<std::pair<std::string, std::string>>* elements)
{
    std::istringstream stream(content);
    std::string line;

    std::regex header{R"^(#{1,6})\s+(.+)$", std::regex::optimize};
    std::regex list_item{R"^(\[*\]\-+)\s+(.+)$", std::regex::optimize};
    std::regex numbered_item{R"^(#\d+\.\s+(.+)$", std::regex::optimize};
    std::regex link{R"\([([^\]]+)\)\[([([^\]]+)\)]\)", std::regex::optimize};

    bool in_paragraph = false;
    std::string paragraph;

    while (std::getline(stream, line)) {
        std::smatch matches;
    }

```

```

// Пустая строка – конец параграфа
if (line.empty()) {
    if (!paragraph.empty()) {
        elements->push_back({"paragraph", paragraph});
        paragraph.clear();
    }
    in_paragraph = false;
    continue;
}

// Заголовок
if (std::regex_match(line, matches, header) && matches.size() > 2) {
    if (!paragraph.empty()) {
        elements->push_back({"paragraph", paragraph});
        paragraph.clear();
    }
    std::string level = std::to_string(matches[1].str().length());
    elements->push_back({"header", "[" + level + " ] " + matches[2].str()});
    in_paragraph = false;
    continue;
}

// Список
if (std::regex_match(line, matches, list_item) && matches.size() > 1) {
    if (!paragraph.empty()) {
        elements->push_back({"paragraph", paragraph});
        paragraph.clear();
    }
    elements->push_back({"list_item", matches[1].str()});
    in_paragraph = false;
    continue;
}

// Нумерованный список
if (std::regex_match(line, matches, numbered_item) && matches.size() > 1) {
    if (!paragraph.empty()) {
        elements->push_back({"paragraph", paragraph});
        paragraph.clear();
    }
}

```

```

        elements->push_back({"list_item", matches[1].str()});
        in_paragraph = false;
        continue;
    }

    // Ссылки в тексте
    std::string processed = line;
    auto link_begin = std::sregex_iterator(line.begin(), line.end(), link);
    auto link_end = std::sregex_iterator();

    for (auto it = link_begin; it != link_end; ++it) {
        if (it->size() > 2) {
            elements->push_back({"link", it->str(1) + " -> " + it->str(2)});
        }
    }

    // Параграф
    if (in_paragraph) {
        paragraph += " " + line;
    } else {
        paragraph = line;
        in_paragraph = true;
    }
}

// Последний параграф
if (!paragraph.empty()) {
    elements->push_back({"paragraph", paragraph});
}
}

```

Parser.h

```

#pragma once

#ifdef MARKDOWN_EXPORTS
#define MD_API __declspec(dllexport)
#else
#define MD_API __declspec(dllimport)
#endif

#include <string>

```

```

#include <list>
extern "C" {
    MD_API void parse_markdown(const char* content,
                                std::list<std::pair<std::string, std::string>>*
                                elements);
}

```

FileProcAsync.cpp

```

#include "FileProcAsync.h"
#include <filesystem>
#include <iostream>
#include <chrono>

namespace fs = std::filesystem;

FileProcAsync::FileProcAsync(const std::string& folder,
                              int threads,
                              std::function<void(const std::string&, std::ofstream&)>
                              func,
                              std::ofstream& outfile)
    : folderPath(folder), numThreads(threads), processorFunc(func),
      outputFile(outfile) {

    if (numThreads <= 0) {
        numThreads = std::thread::hardware_concurrency();
    }
}

void FileProcAsync::start() {
    allFiles = getFilesFromFolder(folderPath);
    totalFiles = allFiles.size();

    readerFuture = std::async(std::launch::async, &FileProcAsync::fileReader, this);

    for (int i = 0; i < numThreads; ++i) {
        workerFutures.push_back(
            std::async(std::launch::async, &FileProcAsync::workerFunction, this)
        );
    }
}

```

```

    }
}

void FileProcAsync::wait() {
    readerFuture.wait();
    finished.store(true);
    cv.notify_all();

    for (auto& w : workerFutures) {
        w.wait();
    }

    std::cout << "Processed " << processedFiles << "/" << totalFiles << " files\n";
}

std::string FileProcAsync::get() {
    std::unique_lock<std::mutex> lock(queueMutex);
    cv.wait(lock, [this] { return !fileQueue.empty() || finished.load(); });

    if (!fileQueue.empty()) {
        std::string content = fileQueue.front();
        fileQueue.pop();
        return content;
    }

    return "";
}

void FileProcAsync::fileReader() {
    for (const auto& filePath : allFiles) {
        std::string content = readFile(filePath);

        std::lock_guard<std::mutex> lock(queueMutex);
        fileQueue.push(content);
        cv.notify_one();
    }
}

void FileProcAsync::workerFunction() {
    while (true) {

```

```

        std::string content = get();
        if (content.empty()) break;

        processFile(content);

        std::lock_guard<std::mutex> lock(progressMutex);
        processedFiles++;

        if (processedFiles % 50 == 0) {
            std::cout << "Progress: " << processedFiles << "/" << totalFiles <<
"\n";
        }
    }
}

void FileProcAsync::processFile(const std::string& content) {
    std::lock_guard<std::mutex> lock(fileMutex);
    processorFunc(content, outputFile);
}

std::vector<std::string> FileProcAsync::getFilesFromFolder(const std::string& path)
{
    std::vector<std::string> files;

    for (const auto& entry : fs::directory_iterator(path)) {
        if (entry.is_regular_file()) {
            std::string ext = entry.path().extension().string();
            std::transform(ext.begin(), ext.end(), ext.begin(), ::tolower);

            if (ext == ".md" || ext == ".markdown") {
                files.push_back(entry.path().string());
            }
        }
    }

    return files;
}

std::string FileProcAsync::readFile(const std::string& path) {
    std::ifstream file(path);
    return std::string((std::istreambuf_iterator<char>(file)),

```

```

        std::istreambuf_iterator<char>());
    }

FileProcAsync.h

#pragma once

#include <future>
#include <vector>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <atomic>
#include <filesystem>
#include <fstream>
#include <functional>
#include <string>

class FileProcAsync {
public:

    FileProcAsync(const std::string& path, int numThreads,
        std::function<void(const std::string&, std::ofstream&)> func,
        std::ofstream& ofile);

    void start();

    void wait();

private:
    std::mutex fileMutex;
    std::mutex progressMutex;
    std::vector<std::string> allFiles;
    size_t totalFiles = 0;
    size_t processedFiles = 0;
    std::chrono::steady_clock::time_point startTime;

    std::string folderPath;

```

```

int numThreads;

std::function<void(const std::string&, std::ofstream&)> processorFunc;
std::ofstream& ofile;


std::queue<std::string> queue;
std::mutex queueMutex;
std::condition_variable cv;
std::atomic<bool> finished;


std::future<void> readerFuture;
std::vector<std::future<void>> workersFutures;


std::string get();


void fileReader();


void workerFunction();


void processFile(const std::string& content);


std::vector<std::string> getFilesFromFolder(const std::string& path);
std::string readFile(const std::string& path);
};

```

DLLLoader.h:

```

#pragma once
#include <string>
#include <list>
#include <windows.h>


class DLLLoader {
private:
    HMODULE dllHandle;

    using ParseFunc = void(*)(const char*, std::list<std::pair<std::string,
std::string>>*>);

    ParseFunc parseFunc;

```

```

public:
    DLLLoader();
    ~DLLLoader();

    bool load(const std::string& dllPath);
    void parse(const std::string& content,
               std::list<std::pair<std::string, std::string>>& elements);
};

```

DLLLoader.cpp:

```

#include "DLLLoader.h"

```

```

DLLLoader::DLLLoader() : dllHandle(nullptr), parseFunc(nullptr) {}
DLLLoader::~DLLLoader() {
    if (dllHandle) {
        FreeLibrary(dllHandle);
    }
}

bool DLLLoader::load(const std::string& dllPath) {
    dllHandle = LoadLibraryA(dllPath.c_str());
    if (!dllHandle) return false;

    parseFunc = reinterpret_cast<ParseFunc>(
        GetProcAddress(dllHandle, "parse_markdown")
    );
    return parseFunc != nullptr;
}

void DLLLoader::parse(const std::string& content,
                     std::list<std::pair<std::string, std::string>>& elements) {
    if (parseFunc) {
        parseFunc(content.c_str(), &elements);
    }
}

```