**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра Вычислительной техники**

**ОТЧЕТ**

**по лабораторной работе №1**

**по дисциплине «Операционные системы»**

**Тема: «Управление файловой системой»**

Студентки гр. 3312 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Поляков А.И.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тимофеев А.В.

Санкт-Петербург

2025

**Цель работы.**

исследовать управление файловой системой с

помощью Win32 API.

**Задание 1.1.**

Управление дисками, каталогами и файлами.

Указания к выполнению.

1. Создайте консольное приложение с меню (каждая выполняемая

функция и/или операция должна быть доступна по отдельному пункту меню),

которое выполняет:

∙ вывод списка дисков (функции Win32 API – GetLogicalDrives,

GetLogicalDriveStrings);

∙ для одного из выбранных дисков вывод информации о диске и размер

свободного пространства (функции Win32 API –

GetDriveType, GetVolumeInformation, GetDiskFreeSpace); ∙

создание и удаление заданных каталогов (функции Win32 API –

CreateDirectory, RemoveDirectory);

∙ создание файлов в новых каталогах (функция Win32 API –

CreateFile);

∙ копирование и перемещение файлов между каталогами с возможностью

выявления попытки работы с файлами, имеющими совпадающие

имена (функции Win32 API – CopyFile, MoveFile, MoveFileEx);

∙ анализ и изменение атрибутов файлов (функции Win32 API –

GetFileAttributes, SetFileAttributes, GetFileInformationByHandle,

GetFileTime, SetFileTime).

**Описание `FileManager.cpp`**

Файл `FileManager.cpp` содержит реализацию класса `FileManager`, который предназначен для управления файлами и директориями в операционной системе Windows. Класс предоставляет функциональность для выполнения различных операций с файловой системой, таких как создание, удаление, копирование и перемещение файлов и директорий, а также для получения и изменения их атрибутов и временных меток.

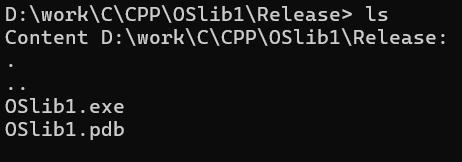
**Основные функции класса**

1. **Конструктор и деструктор**

* Конструктор инициализирует объект класса, устанавливая текущий рабочий каталог.
* Деструктор не выполняет никаких действий, так как в классе нет динамически выделяемых ресурсов.

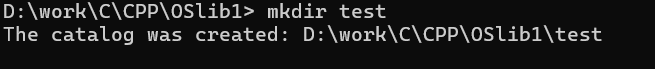
1. **Управление путями**

setPath(const std::wstring& path): Устанавливает текущий рабочий каталог. Поддерживает переход на другой диск, переход в родительскую директорию (`..`) и переход в указанную директорию.



1. **Создание и удаление директорий**

* createDirectory(const std::wstring& dirName): Создает новую директорию в текущем каталоге.

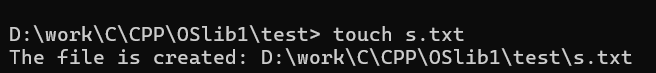


* removeDirectory(const std::wstring& dirName): Удаляет указанную директорию.



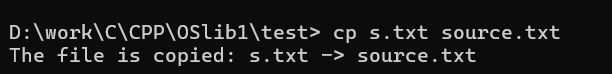
1. **Создание файлов**

createFile(const std::wstring& fileName): Создает новый файл в текущем каталоге.



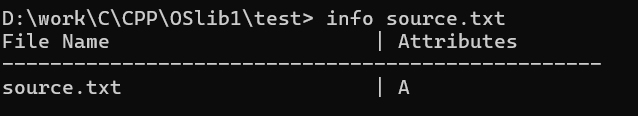
1. **Копирование и перемещение файлов**

* copyFile(const std::wstring& srcFile, const std::wstring& destFile): Копирует файл из исходного пути в целевой.
* moveFile(const std::wstring& srcFile, const std::wstring& destFile): Перемещает или переименовывает файл.



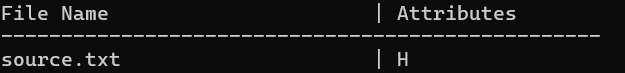
1. **Получение и изменение атрибутов файлов**

* printFileAttributes(const std::wstring& fileName, bool flag): Выводит атрибуты указанного файла.



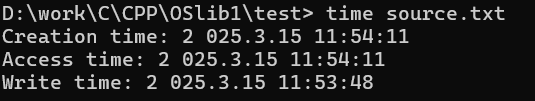
* setFileAttributes(const std::wstring& fileName, const std::wstring& attr): Устанавливает атрибуты для указанного файла.



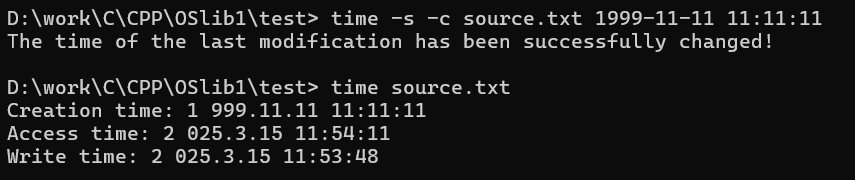


1. **Работа с временными метками файлов**

* showFileTime(const std::wstring& fileName): Выводит временные метки файла (время создания, последнего доступа и последней записи).



* setTimeFile(const std::wstring& fileName, const std::wstring& time, const std::wstring& select): Устанавливает временные метки для указанного файла.



1. **Получение информации о файле**

getFileInfo(const std::wstring& fileName): Выводит подробную информацию о файле, включая его размер, количество ссылок, серийный номер тома и индекс файла.

1. **Список файлов и директорий**

* listFiles(): Выводит список файлов и директорий в текущем каталоге.
* listFilesWithAttributes(const std::wstring& directoryPath): Выводит список файлов и директорий с их атрибутами.

1. **Обработка команд**

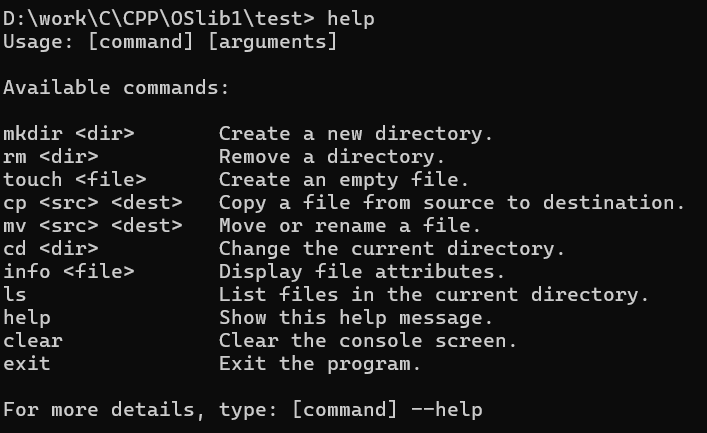
* parseCommand(const std::wstring& command): Разбирает введенную команду на аргументы.
* handleCommand(const std::wstring& command): Обрабатывает команду и вызывает соответствующую функцию.

1. **Интерактивная оболочка**

startShell(): Запускает интерактивную оболочку, которая позволяет пользователю вводить команды и выполнять операции с файловой системой.

1. **Справка**

help(): Выводит справочную информацию о доступных командах.



**Особенности реализации**

Класс использует широкие символы (`wchar\_t`) для поддержки Unicode, что позволяет работать с путями, содержащими не-ASCII символы.

- Для работы с файловой системой используются функции WinAPI, такие как `CreateDirectoryW`, `RemoveDirectoryW`, `CreateFileW`, `CopyFileW`, `MoveFileW`, `GetFileAttributesW`, `SetFileAttributesW`, `GetFileInformationByHandle`, `FindFirstFileW`, `FindNextFileW`, `FindClose`, `SetFileTime`, `GetFileTime`, и другие.

Класс поддерживает обработку ошибок и выводит соответствующие сообщения в случае неудачного выполнения операций.

**Использование**

Класс `FileManager` может быть использован для создания утилит командной строки или интеграции в более крупные приложения, требующие управления файловой системой. Интерактивная оболочка позволяет пользователю легко выполнять операции с файлами и директориями, вводя команды в консоли.

**Описание DiskManager.cpp**

Файл DiskManager.cpp содержит реализацию класса DiskManager, который предназначен для работы с дисками и разделами в операционной системе Windows. Класс предоставляет функциональность для получения информации о доступных дисках, их типах, файловых системах, объеме свободного и занятого пространства, а также для вывода этой информации в удобном формате. Класс также поддерживает интерактивную оболочку для ввода команд.

**Основные функции класса**

1. **Конструктор и деструктор**

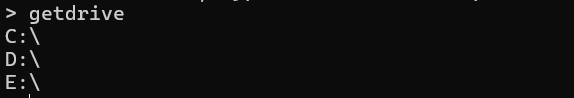
* Конструктор не выполняет никаких действий, так как инициализация не требуется.
* Деструктор также не выполняет никаких действий, так как в классе нет динамически выделяемых ресурсов.

1. **Управление цветом консоли**

SetConsoleColor(int color): Устанавливает цвет текста в консоли. Используется для визуального выделения информации о занятом пространстве на диске.

1. **Получение списка дисков**

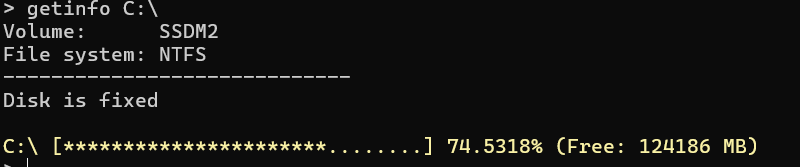
getDrive(): Выводит список всех доступных дисков в системе. Использует функцию GetLogicalDrives для получения информации о подключенных дисках.



1. **Получение информации о диске**

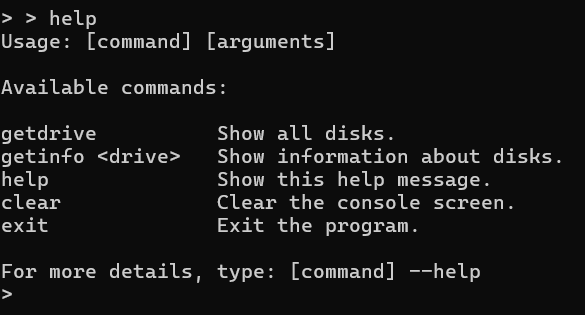
getInfoDrive(std::wstring drive): Выводит подробную информацию о указанном диске, включая:

* Имя тома.
* Тип файловой системы.
* Серийный номер тома.
* Тип диска (съемный, фиксированный, CD-ROM, RAM-диск и т.д.).
* Информацию о свободном и занятом пространстве на диске, включая графическое представление в виде прогресс-бара.



1. **Справка**

help(): Выводит справочную информацию о доступных командах.



1. **Обработка команд**

handleCommand(const std::wstring& command): Разбирает введенную команду и вызывает соответствующую функцию. Поддерживает команды:

* getdrive: Выводит список доступных дисков.
* getinfo <drive>: Выводит информацию о указанном диске.
* clear: Очищает консоль.
* help: Выводит справочную информацию.
* exit: Завершает работу программы.

1. **Интерактивная оболочка**

startShell(): Запускает интерактивную оболочку, которая позволяет пользователю вводить команды и получать информацию о дисках.

**Особенности реализации**

Класс использует широкие символы (wchar\_t) для поддержки Unicode, что позволяет работать с путями, содержащими не-ASCII символы.

Для работы с дисками используются функции WinAPI, такие как GetLogicalDrives, GetVolumeInformationW, GetDriveTypeW, GetDiskFreeSpaceExW.

Класс поддерживает визуализацию занятого пространства на диске с помощью цветного прогресс-бара, что делает вывод информации более наглядным.

Обработка ошибок реализована для случаев, когда информация о диске не может быть получена (например, если диск недоступен или не существует).

**Пример использования**

Пользователь может вводить команды, такие как getdrive, getinfo C:, help, clear, и exit, чтобы управлять выводом информации о дисках и завершать работу программы.

**Вывод**

Класс DiskManager предоставляет удобный инструмент для работы с дисками в Windows, поддерживая получение информации о томах, файловых системах, типах дисков и занятом пространстве. Интерактивная оболочка делает класс простым в использовании, а визуализация данных с помощью цветного прогресс-бара улучшает восприятие информации.

Класс `FileManager` предоставляет удобный и мощный инструмент для работы с файловой системой в Windows, поддерживая широкий спектр операций и обеспечивая обработку ошибок и интерактивный интерфейс для пользователя.

**Текст программы**

*Main.cpp*

#include <iostream>

#include "Menu.h"

int main()

{

std::locale::global(std::locale(""));

std::wcout.imbue(std::locale());

bool interfaceFlag = true;

Menu menu;

while (interfaceFlag)

interfaceFlag = menu.mainMenu();

return 0;

}

Menu.cpp

#include "Menu.h"

Menu::Menu()

{

diskManager = DiskManager();

fileManager = FileManager();

}

void Menu::tableOfContents()

{

std::wcout << L"#=======Select Mode========#" << std::endl;

std::wcout << L"1 - Drive manager" << std::endl;

std::wcout << L"2 - File manager" << std::endl;

std::wcout << L"3 - Exit" << std::endl;

std::wcout << L"#==========================#" << std::endl;

}

bool Menu::mainMenu()

{

int choice;

system("cls");

tableOfContents();

if (firstFlag)

{

std::cin.clear();

while (std::cin.get() != '\n');

firstFlag = true;

}

std::cin >> choice;

system("cls");

switch (choice)

{

case 1:

diskManager.startShell();

break;

case 2:

fileManager.startShell();

break;

case 3:

return false;

break;

default:

break;

}

}

*FileManager.cpp*

#include "FileManager.h"

FileManager::FileManager()

{

wchar\_t buffer[MAX\_PATH];

GetCurrentDirectoryW(MAX\_PATH, buffer);

currentPath = buffer;

}

FileManager::~FileManager() {}

void FileManager::setPath(const std::wstring& path)

{

if (path.length() == 2 && path[1] == L':' && iswalpha(path[0]))

{

std::wstring newDrive = path + L"\\";

if (SetCurrentDirectoryW(newDrive.c\_str()))

currentPath = newDrive;

else

std::wcerr << L"Error: Unable to change to drive " << path << std::endl;

return;

}

if (path == L"..")

{

size\_t pos = currentPath.find\_last\_of(L"\\/");

if (pos != std::wstring::npos)

currentPath = currentPath.substr(0, pos);

if (!SetCurrentDirectoryW(currentPath.c\_str()))

std::wcerr << L"Error: Unable to change directory to " << currentPath << std::endl;

return;

}

std::wstring newPath = currentPath + L"\\" + path;

if (SetCurrentDirectoryW(newPath.c\_str()))

currentPath = newPath;

else

std::wcerr << L"Error: Unable to change directory to " << newPath << std::endl;

}

void FileManager::createDirectory(const std::wstring& dirName)

{

std::wstring fullPath = currentPath + L"\\" + dirName;

if (CreateDirectoryW(fullPath.c\_str(), NULL) || GetLastError() == ERROR\_ALREADY\_EXISTS)

std::wcout << L"The catalog was created: " << fullPath << std::endl;

else

std::wcerr << L"Error: failed to create a catalog " << fullPath << std::endl;

}

void FileManager::removeFile(const std::wstring& fileName)

{

std::wstring fullPath = currentPath + L"\\" + fileName;

if (DeleteFileW(fullPath.c\_str()))

std::wcout << L"The catalog is deleted: " << fullPath << std::endl;

else

std::wcerr << L"Error: failed to remove the catalog " << fullPath << std::endl;

}

void FileManager::removeDirectory(const std::wstring& dirName)

{

std::wstring fullPath = currentPath + L"\\" + dirName;

if (RemoveDirectoryW(fullPath.c\_str()))

std::wcout << L"The catalog is deleted: " << fullPath << std::endl;

else

std::wcerr << L"Error: failed to remove the catalog " << fullPath << std::endl;

}

void FileManager::createFile(const std::wstring& fileName)

{

std::wstring fullPath = currentPath + L"\\" + fileName;

HANDLE hFile = CreateFileW(fullPath.c\_str(), GENERIC\_WRITE, 0, NULL, CREATE\_NEW, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (hFile != INVALID\_HANDLE\_VALUE)

{

CloseHandle(hFile);

std::wcout << L"The file is created: " << fullPath << std::endl;

}

else

std::wcerr << L"Error: failed to create a file " << fullPath << std::endl;

}

void FileManager::copyFile(const std::wstring& srcFile, const std::wstring& destFile)

{

std::wstring srcPath = currentPath + L"\\" + srcFile;

std::wstring destPath;

if (destFile.find(L":\\") != std::wstring::npos || destFile.find(L"\\\\") == 0)

destPath = destFile;

else

destPath = currentPath + L"\\" + destFile;

if (CopyFileW(srcPath.c\_str(), destPath.c\_str(), FALSE))

std::wcout << L"The file is copied: " << srcFile << L" -> " << destFile << std::endl;

else

std::wcerr << L"Error: failed to copy " << srcFile << std::endl;

}

void FileManager::moveFile(const std::wstring& srcFile, const std::wstring& destFile)

{

std::wstring srcPath = currentPath + L"\\" + srcFile;

std::wstring destPath;

if (destFile.find(L":\\") != std::wstring::npos || destFile.find(L"\\\\") == 0)

destPath = destFile;

else

destPath = currentPath + L"\\" + destFile;

if (MoveFileW(srcPath.c\_str(), destPath.c\_str()))

std::wcout << L"The file moved: " << srcFile << L" -> " << destFile << std::endl;

else

std::wcerr << L"Error: failed to move " << srcFile << std::endl;

}

void FileManager::printFileAttributes(const std::wstring& fileName, bool flag)

{

DWORD attributes = GetFileAttributes(fileName.c\_str());

if (attributes == INVALID\_FILE\_ATTRIBUTES)

{

std::wcout << std::setw(30) << std::left << fileName << L" | Error reading attributes" << std::endl;

return;

}

std::wstring attr;

if (attributes & FILE\_ATTRIBUTE\_READONLY) attr += L"R ";

if (attributes & FILE\_ATTRIBUTE\_HIDDEN) attr += L"H ";

if (attributes & FILE\_ATTRIBUTE\_SYSTEM) attr += L"S ";

if (attributes & FILE\_ATTRIBUTE\_DIRECTORY) attr += L"D ";

if (attributes & FILE\_ATTRIBUTE\_ARCHIVE) attr += L"A ";

if (attributes & FILE\_ATTRIBUTE\_DEVICE) attr += L"DEV ";

if (attributes & FILE\_ATTRIBUTE\_NORMAL) attr += L"NORM ";

if (attributes & FILE\_ATTRIBUTE\_TEMPORARY) attr += L"TEMP ";

if (attributes & FILE\_ATTRIBUTE\_SPARSE\_FILE) attr += L"SPARSE ";

if (attributes & FILE\_ATTRIBUTE\_REPARSE\_POINT) attr += L"REPARSE ";

if (attributes & FILE\_ATTRIBUTE\_COMPRESSED) attr += L"COMP ";

if (attributes & FILE\_ATTRIBUTE\_OFFLINE) attr += L"OFFLINE ";

if (attributes & FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED) attr += L"NOINDEX ";

if (attributes & FILE\_ATTRIBUTE\_ENCRYPTED) attr += L"ENC ";

if (flag)

{

std::wcout << std::setw(30) << std::left << L"File Name" << L" | Attributes" << std::endl;

std::wcout << std::wstring(50, L'-') << std::endl;

}

std::wcout << std::setw(30) << std::left << fileName << L" | " << attr << std::endl;

}

void FileManager::listFilesWithAttributes(const std::wstring& directoryPath)

{

bool flag = true;

WIN32\_FIND\_DATA findFileData;

HANDLE hFind = FindFirstFile((directoryPath + L"\\\*").c\_str(), &findFileData);

if (hFind == INVALID\_HANDLE\_VALUE)

{

std::wcerr << L"Error opening directory: " << directoryPath << std::endl;

return;

}

do

{

printFileAttributes(directoryPath + L"\\" + findFileData.cFileName, flag);

flag = false;

}

while (FindNextFile(hFind, &findFileData) != 0);

FindClose(hFind);

}

void FileManager::getFileInfo(const std::wstring& fileName)

{

std::wstring fullPath = currentPath + L"\\" + fileName;

HANDLE hFile = CreateFileW(fullPath.c\_str(), GENERIC\_READ, FILE\_SHARE\_READ, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (hFile == INVALID\_HANDLE\_VALUE)

{

std::wcerr << L"Error: Unable to open file " << fullPath << std::endl;

return;

}

BY\_HANDLE\_FILE\_INFORMATION fileInfo;

if (GetFileInformationByHandle(hFile, &fileInfo))

{

std::wcout << L"File Information for: " << fileName << std::endl;

std::wcout << L"File size: " << ((static\_cast<ULONGLONG>(fileInfo.nFileSizeHigh) << 32) | fileInfo.nFileSizeLow) << L" bytes" << std::endl;

std::wcout << L"Number of links: " << fileInfo.nNumberOfLinks << std::endl;

std::wcout << L"Volume serial number: " << fileInfo.dwVolumeSerialNumber << std::endl;

std::wcout << L"File index: " << fileInfo.nFileIndexHigh << L"-" << fileInfo.nFileIndexLow << std::endl;

}

else

std::wcerr << L"Error: Unable to retrieve file information." << std::endl;

CloseHandle(hFile);

}

void FileManager::help()

{

std::wcout << L"Usage: [command] [arguments]" << std::endl << std::endl;

std::wcout << L"Available commands:" << std::endl << std::endl;

std::wcout << L"mkdir <dir> Create a new directory." << std::endl;

std::wcout << L"rm <dir> Remove a directory." << std::endl;

std::wcout << L"touch <file> Create an empty file." << std::endl;

std::wcout << L"cp <src> <dest> Copy a file from source to destination." << std::endl;

std::wcout << L"mv <src> <dest> Move or rename a file." << std::endl;

std::wcout << L"cd <dir> Change the current directory." << std::endl;

std::wcout << L"info <file> Display file attributes." << std::endl;

std::wcout << L"ls List files in the current directory." << std::endl;

std::wcout << L"help Show this help message." << std::endl;

std::wcout << L"clear Clear the console screen." << std::endl;

std::wcout << L"exit Exit the program." << std::endl << std::endl;

std::wcout << L"For more details, type: [command] --help" << std::endl;

}

void FileManager::listFiles()

{

std::wstring searchPath = currentPath + L"\\\*";

WIN32\_FIND\_DATAW findFileData;

HANDLE hFind = FindFirstFileW(searchPath.c\_str(), &findFileData);

if (hFind == INVALID\_HANDLE\_VALUE)

{

std::wcerr << L"Error: failed to open the catalog " << currentPath << std::endl;

return;

}

std::wcout << L"Content " << currentPath << L":\n";

do

std::wcout << findFileData.cFileName << std::endl;

while (FindNextFileW(hFind, &findFileData));

FindClose(hFind);

}

void FileManager::printFileTime(const FILETIME& fileTime)

{

SYSTEMTIME systemTime;

FileTimeToSystemTime(&fileTime, &systemTime);

std::wcout << systemTime.wYear << L"." << systemTime.wMonth << L"." << systemTime.wDay << L" " << systemTime.wHour << L":" << systemTime.wMinute << L":" << systemTime.wSecond << std::endl;

}

void FileManager::showFileTime(const std::wstring& fileName)

{

std::wstring searchPath = currentPath + L"\\" + fileName;

HANDLE hFile = CreateFile(searchPath.c\_str(), GENERIC\_READ, FILE\_SHARE\_READ, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

FILETIME cretionFile, accessTime, writeTime;

if (GetFileTime(hFile, &cretionFile, &accessTime, &writeTime))

{

std::wcout << L"Creation time: ";

printFileTime(cretionFile);

std::wcout << L"Access time: ";

printFileTime(accessTime);

std::wcout << L"Write time: ";

printFileTime(writeTime);

}

else

std::wcerr << L"Error\n";

CloseHandle(hFile);

}

void FileManager::setTimeFile(const std::wstring& fileName, const std::wstring& time, const std::wstring& select)

{

std::wstring filename = currentPath + L"\\" + fileName;

HANDLE hFile = CreateFile(filename.c\_str(), GENERIC\_WRITE, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (hFile == INVALID\_HANDLE\_VALUE)

{

std::cerr << "File opening error\n";

return;

}

FILETIME ft;

SYSTEMTIME st = {};

std::wstringstream ss(time);

wchar\_t delimiter;

ss >> st.wYear >> delimiter >> st.wMonth >> delimiter >> st.wDay >> st.wHour >> delimiter >> st.wMinute >> delimiter >> st.wSecond;

st.wMilliseconds = 0;

if (!SystemTimeToFileTime(&st, &ft))

{

std::cerr << "Time enveloping error\n";

CloseHandle(hFile);

return;

}

if (select == L"-w")

{

if (SetFileTime(hFile, NULL, NULL, &ft))

std::cout << "The time of the last modification has been successfully changed!\n";

else

std::cerr << "Time changes error\n";

}

else if (select == L"-a")

{

if (SetFileTime(hFile, NULL, &ft, NULL))

std::cout << "The time of the last modification has been successfully changed!\n";

else

std::cerr << "Time changes error\n";

}

else

{

if (SetFileTime(hFile, &ft, NULL, NULL))

std::cout << "The time of the last modification has been successfully changed!\n";

else

std::cerr << "Time changes error\n";

}

CloseHandle(hFile);

}

void FileManager::setFileAttributes(const std::wstring& fileName, const std::wstring& attr)

{

std::unordered\_map<std::wstring, DWORD> attrMap =

{

{L"R", FILE\_ATTRIBUTE\_READONLY},

{L"H", FILE\_ATTRIBUTE\_HIDDEN},

{L"S", FILE\_ATTRIBUTE\_SYSTEM},

{L"D", FILE\_ATTRIBUTE\_DIRECTORY},

{L"A", FILE\_ATTRIBUTE\_ARCHIVE},

{L"DEV", FILE\_ATTRIBUTE\_DEVICE},

{L"NORM", FILE\_ATTRIBUTE\_NORMAL},

{L"TEMP", FILE\_ATTRIBUTE\_TEMPORARY},

{L"SPARSE", FILE\_ATTRIBUTE\_SPARSE\_FILE},

{L"REPARSE", FILE\_ATTRIBUTE\_REPARSE\_POINT},

{L"COMP", FILE\_ATTRIBUTE\_COMPRESSED},

{L"OFFLINE", FILE\_ATTRIBUTE\_OFFLINE},

{L"NOINDEX", FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED},

{L"ENC", FILE\_ATTRIBUTE\_ENCRYPTED}

};

DWORD fileAttributes = 0;

size\_t start = 0, end = 0;

while ((end = attr.find(L' ', start)) != std::wstring::npos)

{

std::wstring token = attr.substr(start, end - start);

if (attrMap.count(token))

fileAttributes |= attrMap[token];

else

throw std::invalid\_argument("Unknown file attribute: " + std::string(token.begin(), token.end()));

start = end + 1;

}

if (start < attr.size())

{

std::wstring token = attr.substr(start);

if (attrMap.count(token))

fileAttributes |= attrMap[token];

else

throw std::invalid\_argument("Unknown file attribute: " + std::string(token.begin(), token.end()));

}

if (!SetFileAttributesW(fileName.c\_str(), fileAttributes))

throw std::runtime\_error("Failed to set the attributes of the file");

}

std::vector<std::wstring> FileManager::parseCommand(const std::wstring& command)

{

std::vector<std::wstring> args;

std::wstringstream ss(command);

std::wstring arg;

bool inQuotes = false;

wchar\_t quoteChar = 0;

std::wstring currentArg;

while (ss)

{

wchar\_t ch = ss.get();

if (ss.eof())

break;

if ((ch == L'"' || ch == L'\'') && !inQuotes)

{

inQuotes = true;

quoteChar = ch;

}

else if (ch == quoteChar && inQuotes)

{

inQuotes = false;

args.push\_back(currentArg);

currentArg.clear();

}

else if (ch == L' ' && !inQuotes)

{

if (!currentArg.empty())

{

args.push\_back(currentArg);

currentArg.clear();

}

}

else

currentArg += ch;

}

if (!currentArg.empty())

args.push\_back(currentArg);

return args;

}

void FileManager::handleCommand(const std::wstring& command)

{

std::vector<std::wstring> args = parseCommand(command);

if (args.empty())

return;

if (args[0] == L"mkdir" && args.size() == 2)

createDirectory(args[1]);

else if (args[0] == L"rm" && args.size() == 2)

removeFile(args[1]);

else if (args[0] == L"rmdir" && args.size() == 2)

removeDirectory(args[1]);

else if (args[0] == L"touch" && args.size() == 2)

createFile(args[1]);

else if (args[0] == L"cp" && args.size() == 3)

copyFile(args[1], args[2]);

else if (args[0] == L"mv" && args.size() == 3)

moveFile(args[1], args[2]);

else if (args[0] == L"cd" && args.size() == 2)

setPath(args[1]);

else if (args[0] == L"info" && args.size() == 2)

printFileAttributes(args[1]);

else if (args[0] == L"info" && args[1] == L"-a" && args.size() == 3)

listFilesWithAttributes(args[2]);

else if (args[0] == L"info" && args[1] == L"-f" && args.size() == 3)

getFileInfo(args[2]);

else if (args[0] == L"info" && args[1] == L"-s" && args.size() == 4)

setFileAttributes(args[2], args[3]);

else if (args[0] == L"time" && args.size() == 2)

showFileTime(args[1]);

else if (args[0] == L"time" && args[1] == L"-s" && (args[2] == L"-c" || args[2] == L"-a" || args[2] == L"-w") && args.size() == 6)

{

std::wstring timeString = args[4] + L" " + args[5];

setTimeFile(args[3], timeString, args[2]);

}

else if (args[0] == L"ls")

listFiles();

else if (args[0] == L"help")

help();

else if (args[0] == L"clear")

system("cls");

else if (args[0] == L"exit")

loopFlag = false;

else

std::wcout << L"Unknown command! " << L"For info enter \"help\" " << std::endl;

}

void FileManager::startShell()

{

std::wstring input;

while (loopFlag)

{

std::wcout << L"\n" << currentPath << L"> ";

std::getline(std::wcin, input);

handleCommand(input);

}

loopFlag = true;

}

DiskManager.cpp

#include "DiskManager.h"

DiskManager::DiskManager() {}

void DiskManager::SetConsoleColor(int color)

{

HANDLE hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(hConsole, color);

}

void DiskManager::getDrive()

{

DWORD drives = GetLogicalDrives();

for (char letter = 'A'; letter <= 'Z'; ++letter)

if (drives & (1 << (letter - 'A')))

std::cout << letter << ":\\" << std::endl;

}

void DiskManager::help()

{

std::wcout << L"Usage: [command] [arguments]" << std::endl << std::endl;

std::wcout << L"Available commands:" << std::endl << std::endl;

std::wcout << L"getdrive Show all disks." << std::endl;

std::wcout << L"getinfo <drive> Show information about disks." << std::endl;

std::wcout << L"help Show this help message." << std::endl;

std::wcout << L"clear Clear the console screen." << std::endl;

std::wcout << L"exit Exit the program." << std::endl << std::endl;

std::wcout << L"For more details, type: [command] --help" << std::endl;

}

void DiskManager::getInfoDrive(std::wstring drive)

{

std::wstring wdrive = std::wstring(drive.begin(), drive.end());

wchar\_t volumeName[MAX\_PATH] = { 0 };

wchar\_t fileSystem[MAX\_PATH] = { 0 };

DWORD serialNumber = 0, maxComponentLen = 0, fileSystemFlags = 0;

ULARGE\_INTEGER freeBytesAvailable, totalBytes, totalFreeBytes;

if (GetVolumeInformationW(wdrive.c\_str(), volumeName, MAX\_PATH, &serialNumber, &maxComponentLen, &fileSystemFlags, fileSystem, MAX\_PATH))

std::wcout << L"Volume: " << volumeName << L"\nFile system: " << fileSystem << std::endl;

else

std::wcerr << "Failed to get volume information for drive " << drive << std::endl;

std::wcout << L"-----------------------------" << std::endl;

UINT driveType = GetDriveTypeW(wdrive.c\_str());

if (driveType == DRIVE\_NO\_ROOT\_DIR)

{

std::wcerr << "Error: Invalid drive " << drive << std::endl;

return;

}

else if (driveType == DRIVE\_UNKNOWN)

{

std::wcerr << "Error: Unknow drive " << drive << std::endl;

return;

}

else

{

if (driveType == DRIVE\_REMOVABLE)

std::wcout << L"Disk is REMOVABLE" << std::endl;

if (driveType == DRIVE\_FIXED)

std::wcout << L"Disk is fixed" << std::endl;

if (driveType == DRIVE\_REMOTE)

std::wcout << L"Disk is remote" << std::endl;

if (driveType == DRIVE\_CDROM)

std::wcout << L"Disk is CD-ROM" << std::endl;

if (driveType == DRIVE\_RAMDISK)

std::wcout << L"Disk is RAM" << std::endl;

}

std::wcout << std::endl;

if (GetDiskFreeSpaceExW(wdrive.c\_str(), &freeBytesAvailable, &totalBytes, &totalFreeBytes))

{

double usedSpace = (totalBytes.QuadPart - totalFreeBytes.QuadPart) \* 100.0 / totalBytes.QuadPart;

int barWidth = 30;

int filled = static\_cast<int>(barWidth \* usedSpace / 100);

if (usedSpace < 50)

SetConsoleColor(10);

else if (usedSpace < 80)

SetConsoleColor(14);

else

SetConsoleColor(12);

std::wcout << wdrive << L" [";

for (int i = 0; i < barWidth; i++)

{

if (i < filled)

std::wcout << L"\*";

else

std::wcout << L".";

}

std::wcout << L"] " << usedSpace << L"% (" << L"Free: " << (totalFreeBytes.QuadPart / (1024 \* 1024)) << L" MB)" << std::endl;

SetConsoleColor(7);

}

else

std::wcerr << "Failed to get disk space info for drive " << drive << std::endl;

}

void DiskManager::handleCommand(const std::wstring& command)

{

std::wstringstream ss(command);

std::vector<std::wstring> args;

std::wstring arg;

while (ss >> arg)

args.push\_back(arg);

if (args.empty())

return;

if (args[0] == L"getdrive")

getDrive();

else if (args[0] == L"getinfo" && args.size() == 2)

getInfoDrive(args[1]);

else if (args[0] == L"clear")

system("cls");

else if (args[0] == L"help")

help();

else if (args[0] == L"exit")

loopFlag = false;

else

std::wcout << L"Unknown command!" << L"For info enter \"help\" " << std::endl;

}

void DiskManager::startShell()

{

std::wstring input;

while (loopFlag)

{

std::wcout << L"> ";

std::getline(std::wcin, input);

handleCommand(input);

}

loopFlag = true;

}

DiskManager::~DiskManager() {}