1. МИНОБРНАУКИ РОССИИ
2. САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
3. ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
4. «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
5. Кафедра Вычислительной техники

ОТЧЁТ

по лабораторной работе №1

1. по дисциплине «Операционные системы»
2. Тема: Управление файловой системой

|  |  |  |
| --- | --- | --- |
| Студент гр. 2307 |  | Подберёзский А.Д. |
| Преподаватель |  | Тимофеев А.В. |

Санкт-Петербург

2024

# 1. Введение

Тема работы: Управление файловой системой.

Цель работы: Исследование управления файловой системой с помощью Win32 API.

Указания к выполнению

Задание 1.1. Управление дисками, каталогами и файлами.

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).

2. Запустите приложение и проверьте его работоспособность на нескольких наборах вводимых данных. Запротоколируйте результаты в отчёт. Дайте свои комментарии в отчёте относительно выполнения функций Win32 API.

3. Подготовьте итоговый отчёт с развернутыми выводами по заданию.

Задание 1.2. Копирование файла с помощью операций перекрывающегося ввода-вывода. Приложение должно копировать существующий файл в новый файл, «одновременно» выполняя n перекрывающихся операций ввода-вывода (механизм APC) блоками данных кратными размеру кластера. Указания к выполнению.

1. Создайте консольное приложение, которое выполняет:

 открытие/создание файлов (функция Win32 API – CreateFile, обязательно использовать флаги FILE\_FLAG\_NO\_BUFFERING и FILE\_FLAG\_OVERLAPPED);

 файловый ввод-вывод (функции Win32 API – ReadFileEx, WriteFileEx) блоками кратными размеру кластера;

 ожидание срабатывания вызова функции завершения (функция Win32 API – SleepEx);

 измерение продолжительности выполнения операции копирования файла (функция Win32 API – TimeGetTime).

2. Запустите приложение и проверьте его работоспособность на копировании файлов разного размера для ситуации с перекрывающимся выполнением одной операции ввода и одной операции вывода (для сравнения файлов используйте консольную команду FC). Выполните эксперимент для разного размера копируемых блоков, постройте график зависимости скорости копирования от размера блока данных. Определите оптимальный размер блока данных, при котором скорость копирования наибольшая. Запротоколируйте результаты в отчёт. Дайте свои комментарии в отчете относительно выполнения функций Win32 API.

3. Произведите замеры времени выполнения приложения для разного числа перекрывающихся операций ввода и вывода (1, 2, 4, 8, 12, 16), не забывая проверять работоспособность приложения (консольная команда FC). По результатам измерений постройте график зависимости и определите число перекрывающихся операций ввода и вывода, при котором достигается наибольшая скорость копирования файла. Запротоколируйте результаты в отчёт.

# 2. Управление дисками, каталогами и файлами

## 2.1. Вывод списка дисков

Реализация вывода списка дисков с помощью функции GetDList();

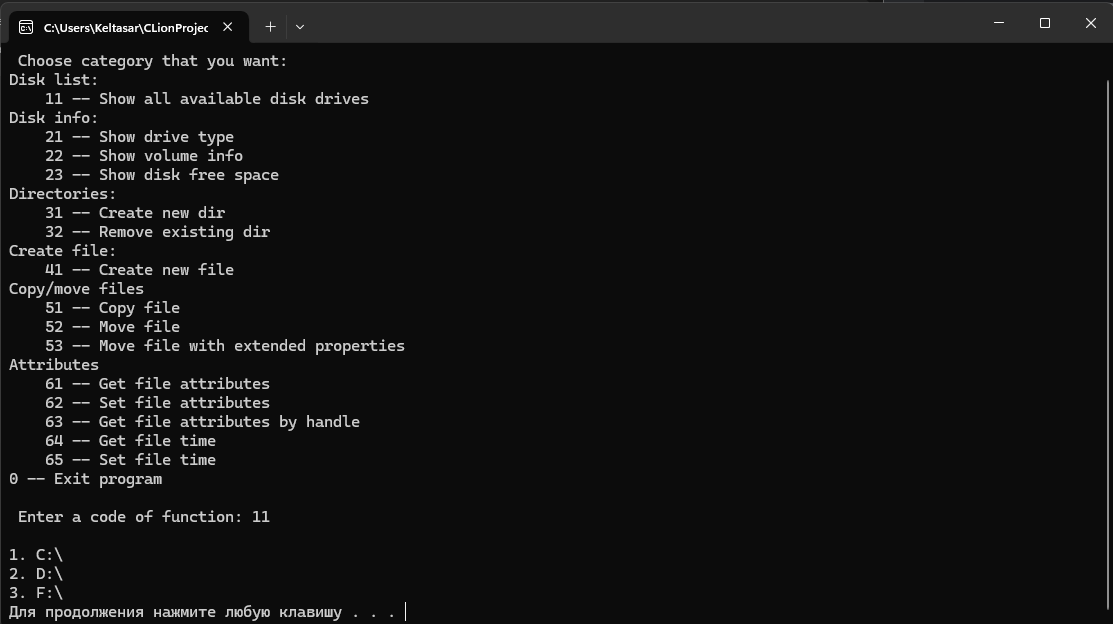


Рисунок 1: Вывод списка дисков

## 2.2. Вывод информации о диске и размер свободного пространства

Вывод основной информации о дисках (в том числе свободное пространство, их системные данные и их системные флаги) с использованием функций GetDriveType(), GetVolumeInformation() и GetDiskFreeSpace().

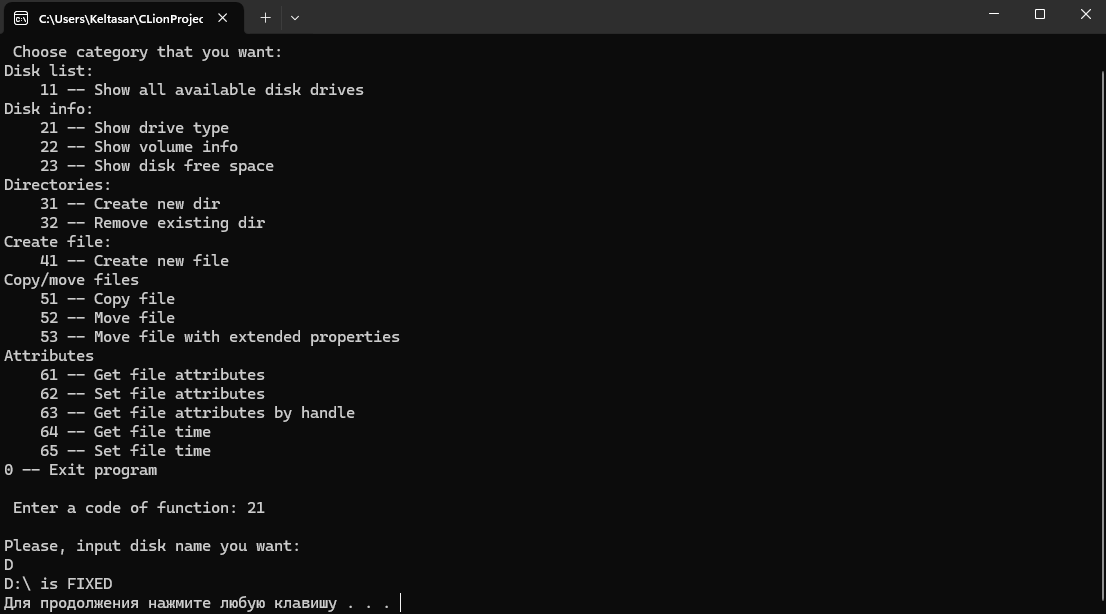


Рисунок 2: Вывод типа диска

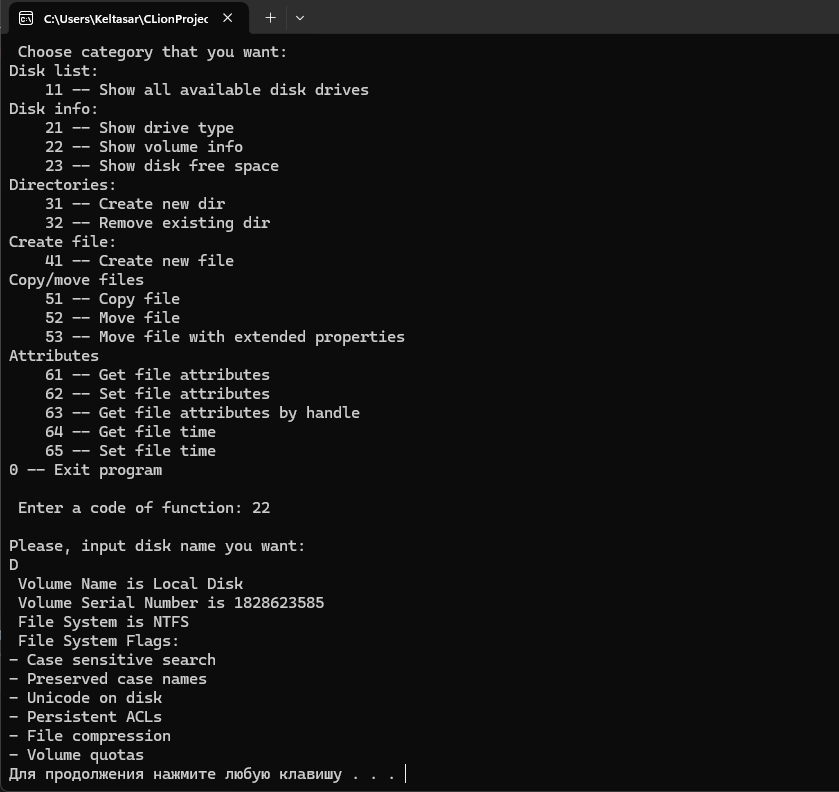


Рисунок 3: Вывод информации о диске

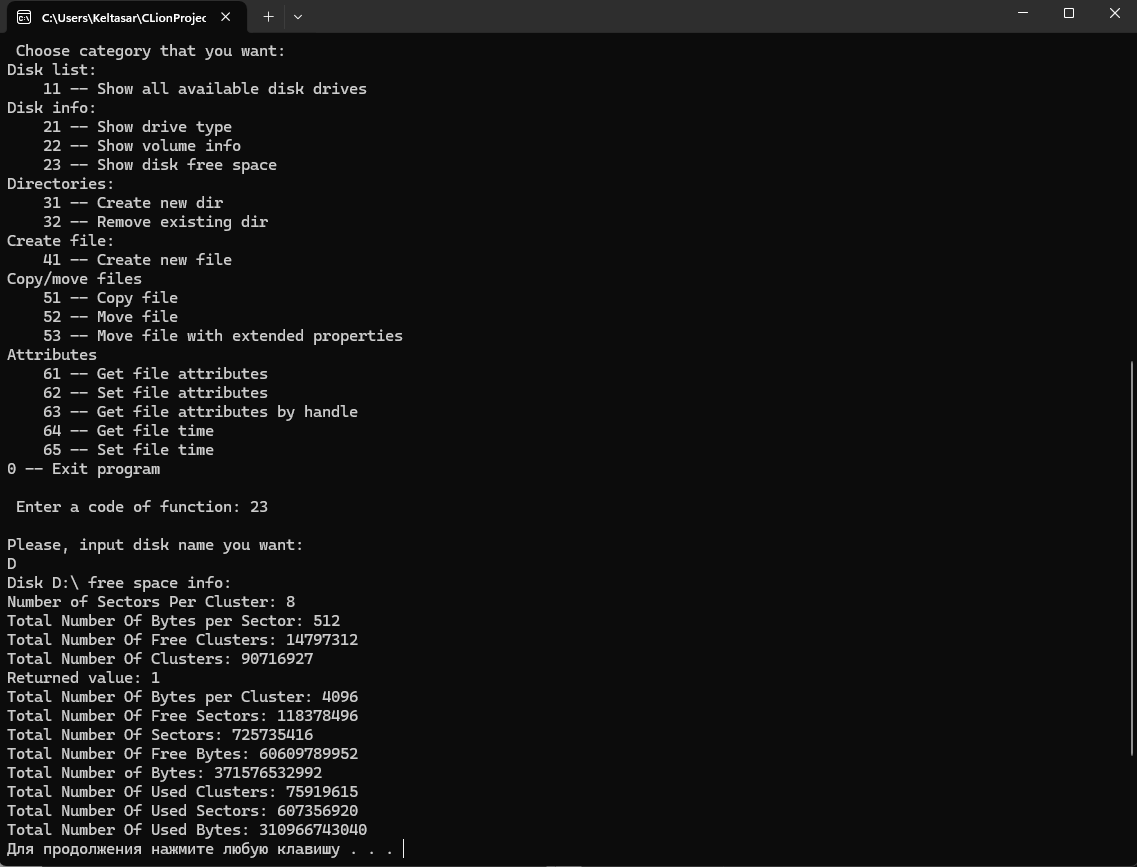


Рисунок 4: Вывод информации о пространстве на диске

## 2.3. Создание и удаление заданных каталогов

Реализация создания и удаления заданных каталогов с помощью функций CreateDirectory() и RemoveDirectory().

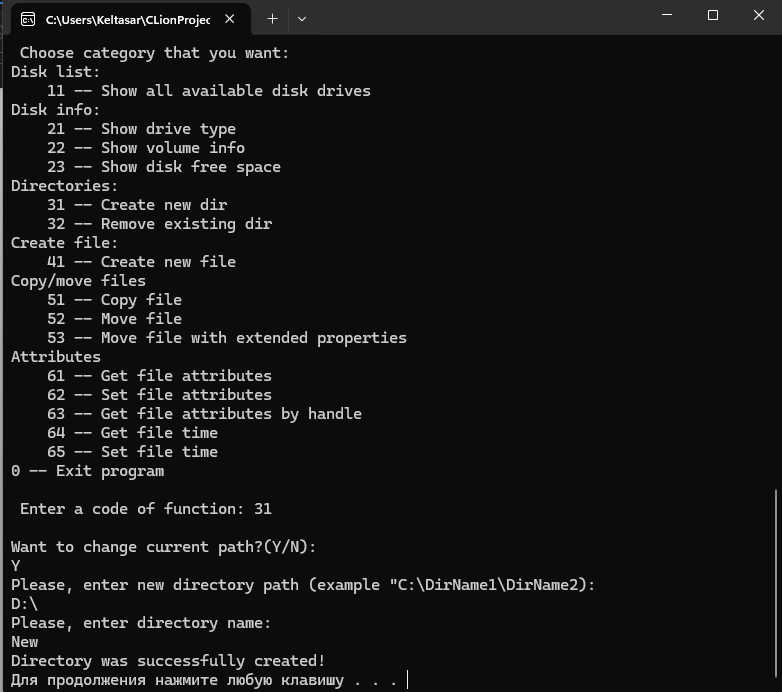


Рисунок 5: Создание каталога

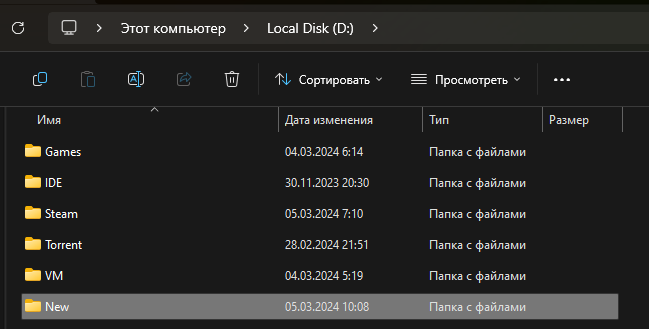


Рисунок 6: Директория каталога

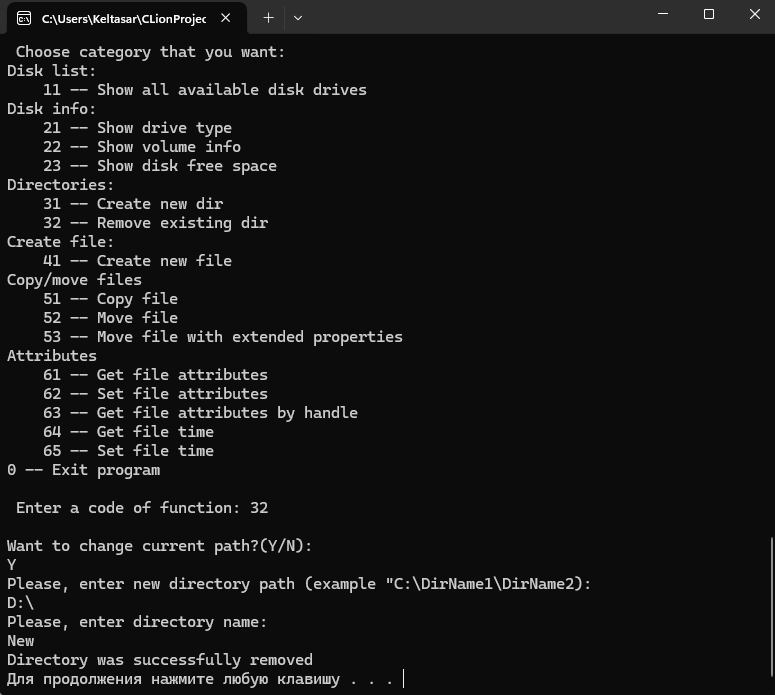


Рисунок 7: Удаление каталога

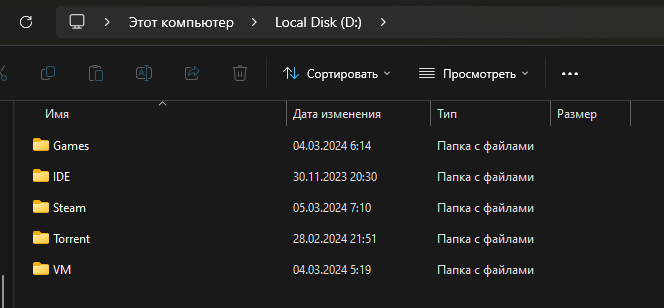


Рисунок 10: Директория каталога

## 2.4. Создание файлов в новых каталогах

Реализация создания файлов в новых каталогах с возможностью использовать флаги для этой функции (например, для открытия) с помощью функции CreateFile().

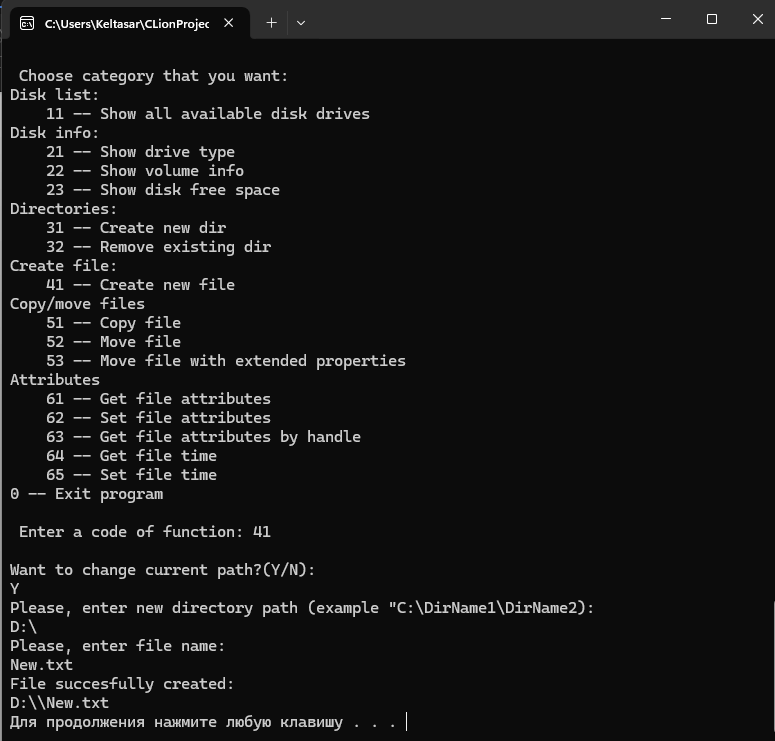


Рисунок 11: Создание файла

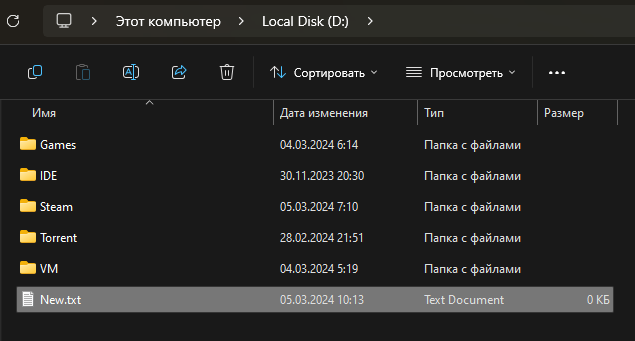
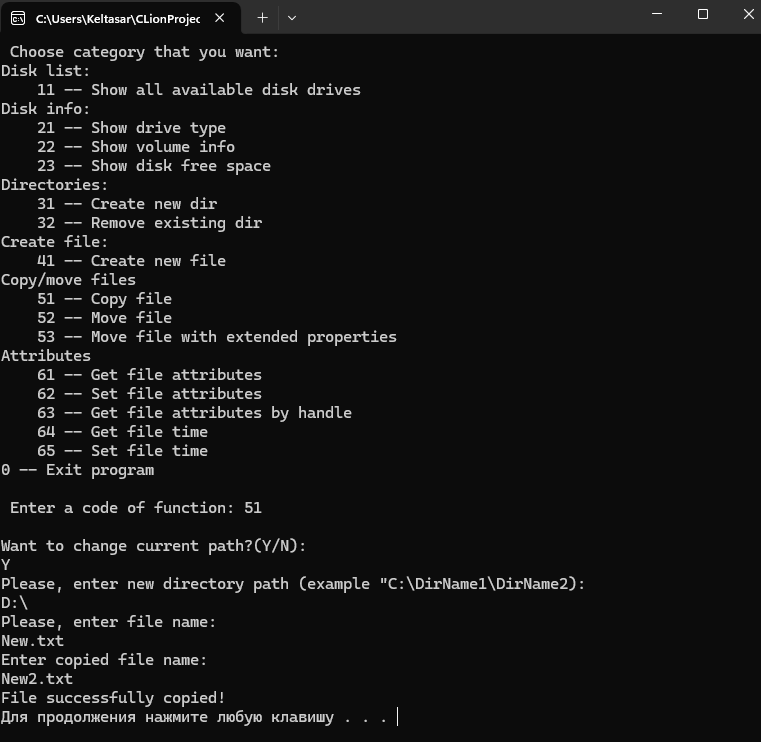


Рисунок 12: Директория файла

## 2.5. Копирование и перемещение файлов между каталогами

Реализация копирования и перемещения файлов между каталогами (в том числе с возможностью выбрать, как поведёт себя консольное приложение при возникновении файлов с одинаковым именем) с помощью функций CopyFile(), MoveFile() и MoveFileEx().

Рисунок 13: Копирование файла

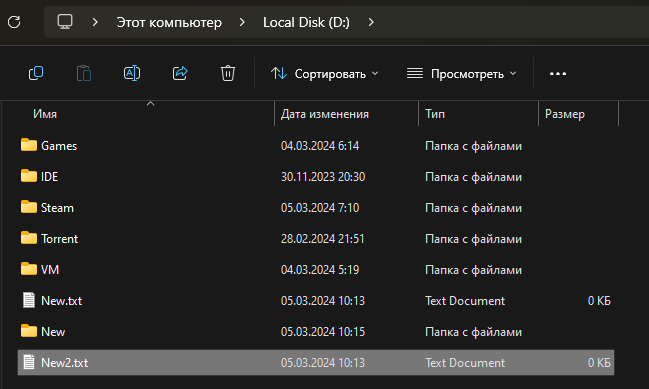
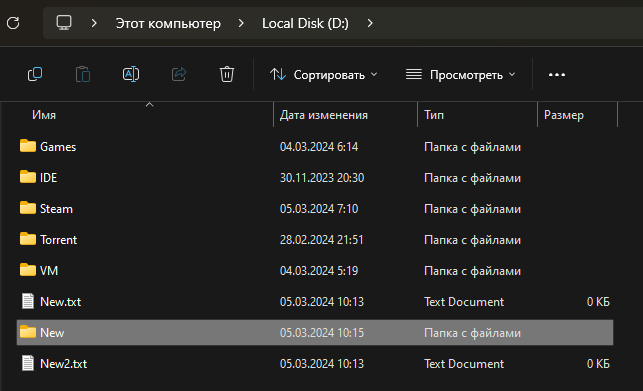
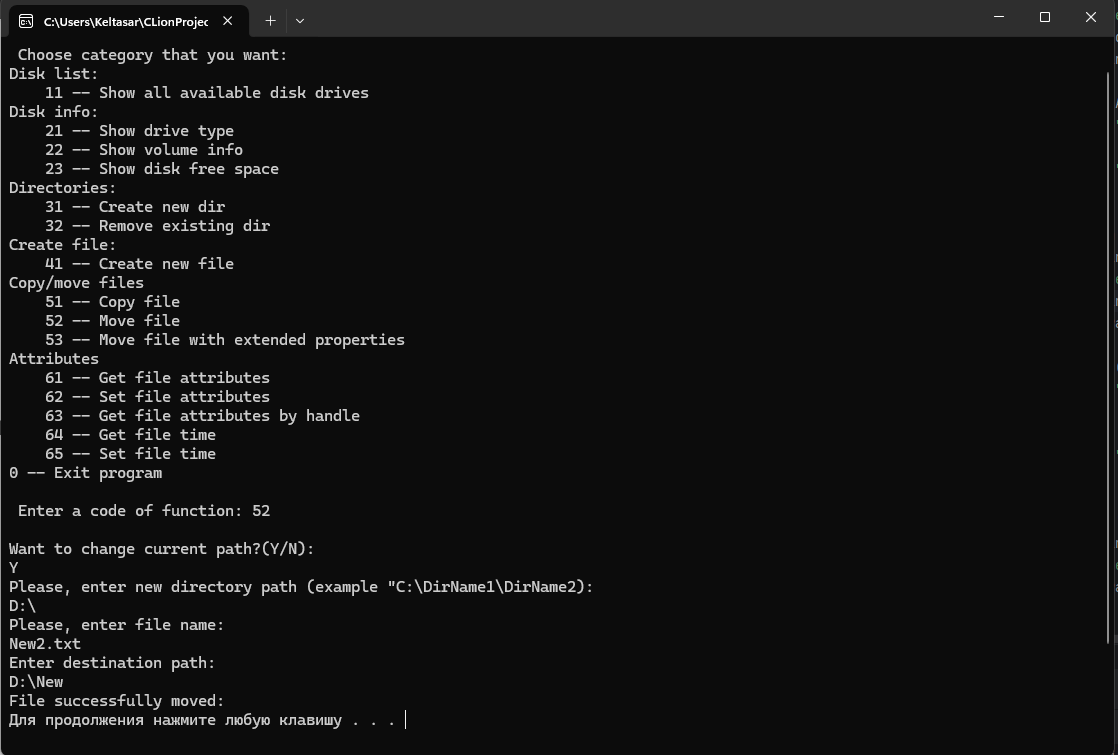
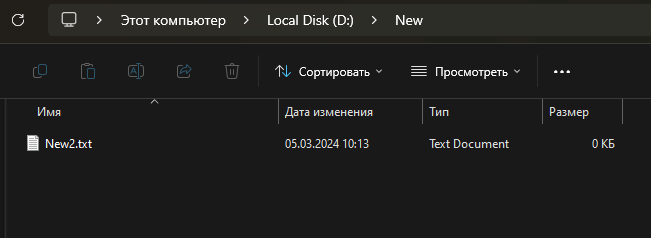


Рисунок 14: Директория файла

Рисунок 15: Директория файла до перемещения

Рисунок 16: Перемещение файла

Рисунок 17: Директория файла после перемещения

## 2.6. Анализ и изменение атрибутов файлов

Реализация анализа и изменения атрибутов файлов, а также их временных данных, получения данных с помощью дескриптора с помощью функций GetFileAttributes(), SetFileAttributes(), GetFileInformationByHandle(), GetFileTime() и SetFileTime().

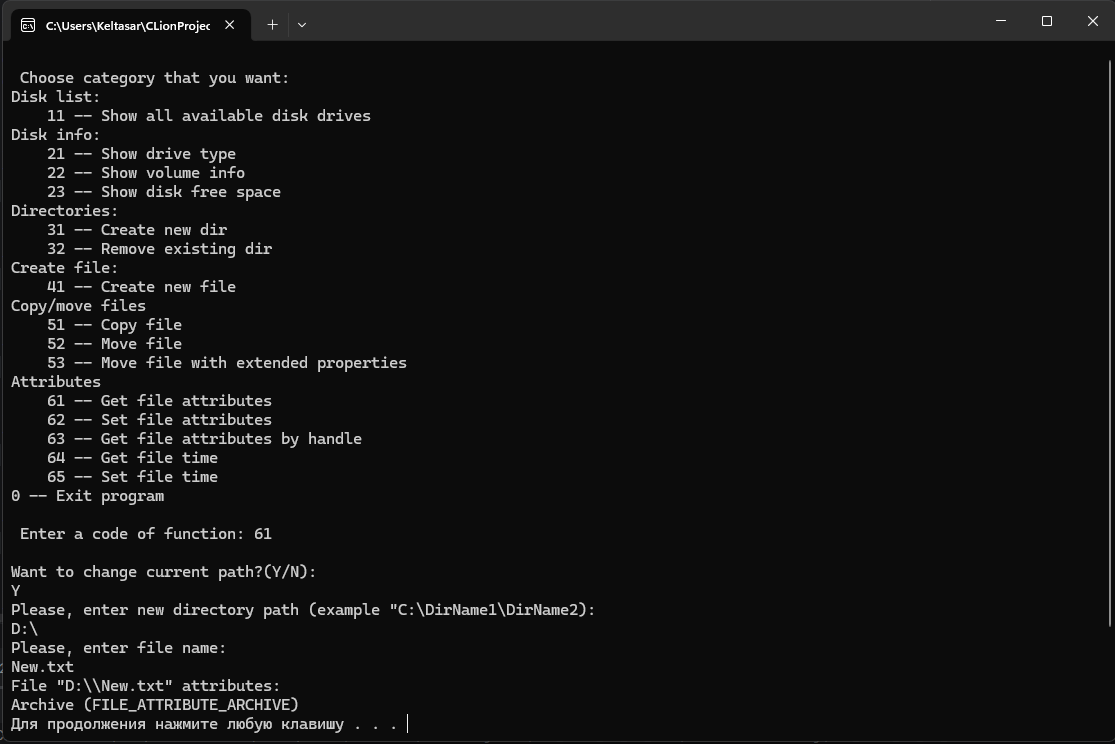


Рисунок 18: Вывод атрибутов файла

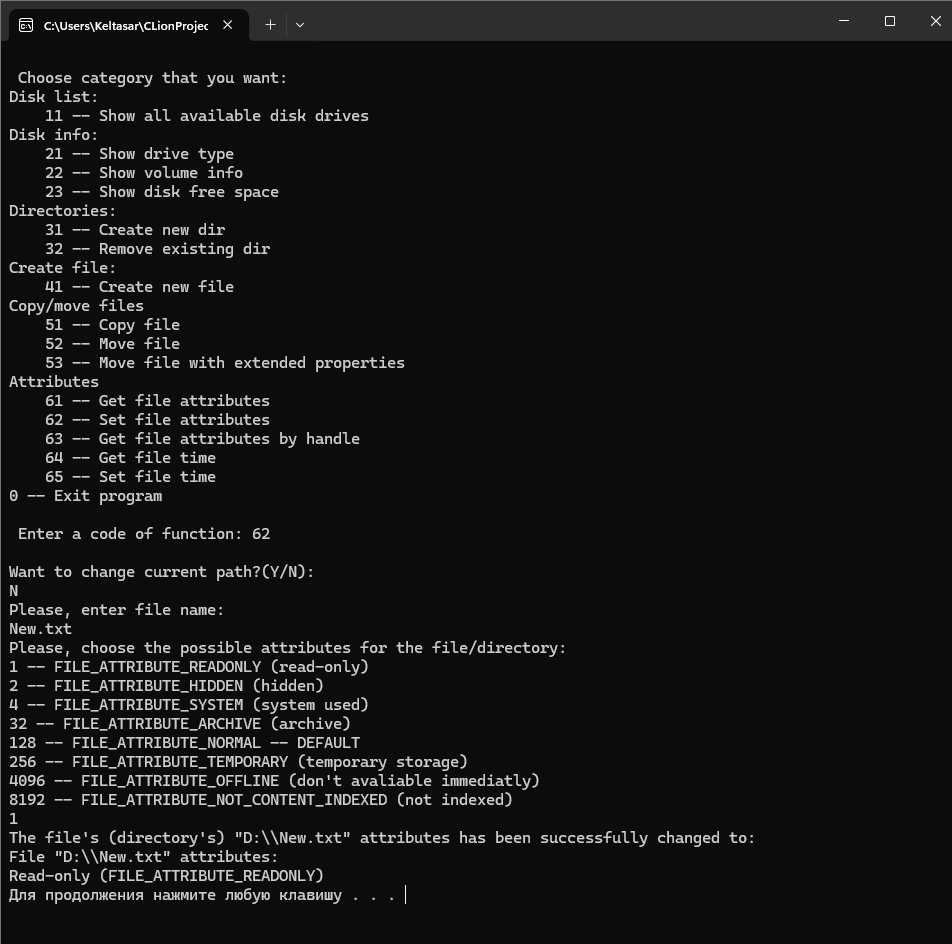


Рисунок 19: Изменение атрибутов файла

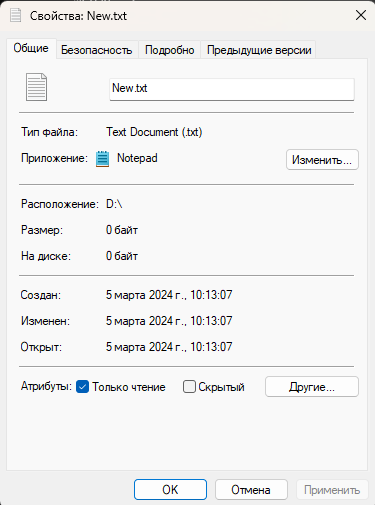


Рисунок 20: Изменённый атрибут файла в свойствах

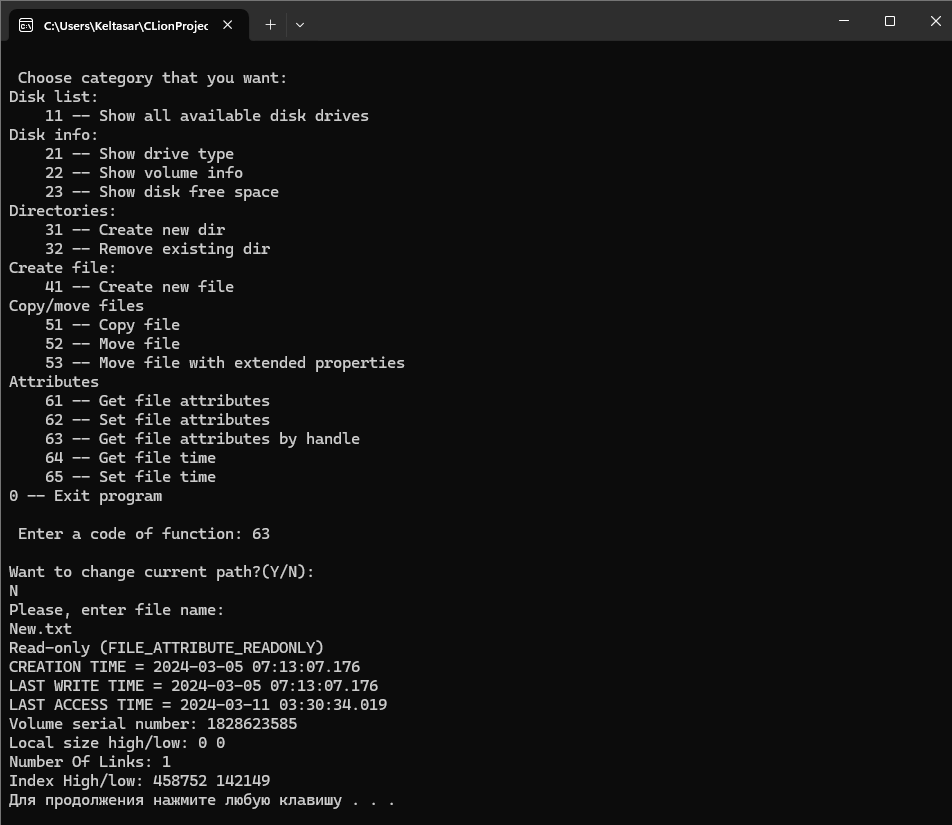


Рисунок 21: Вывод атрибутов файла по дескриптору

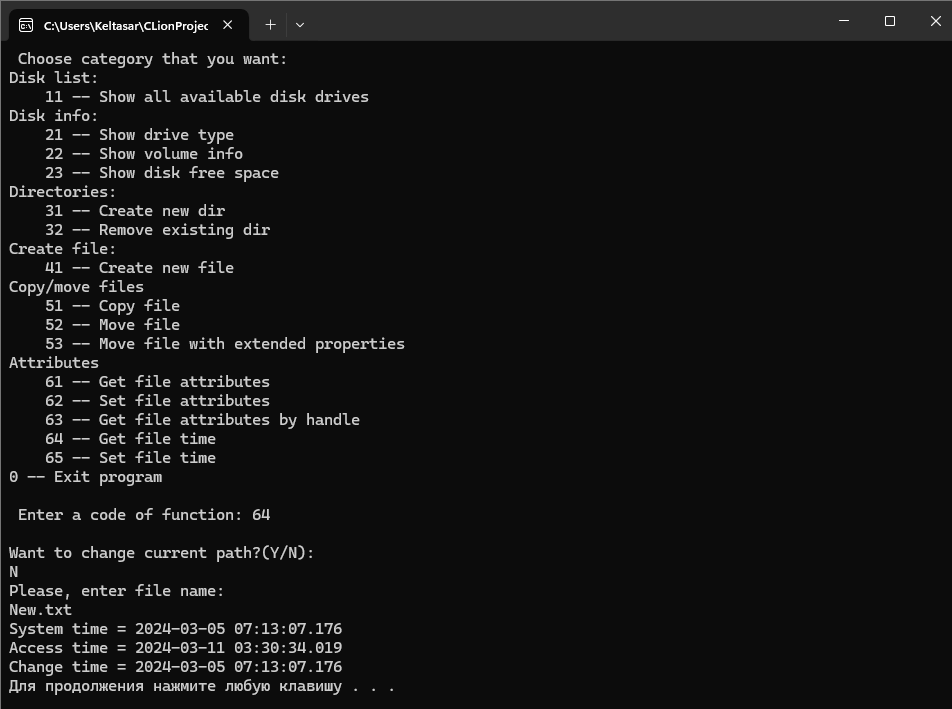


Рисунок 22: Вывод значений времени файла

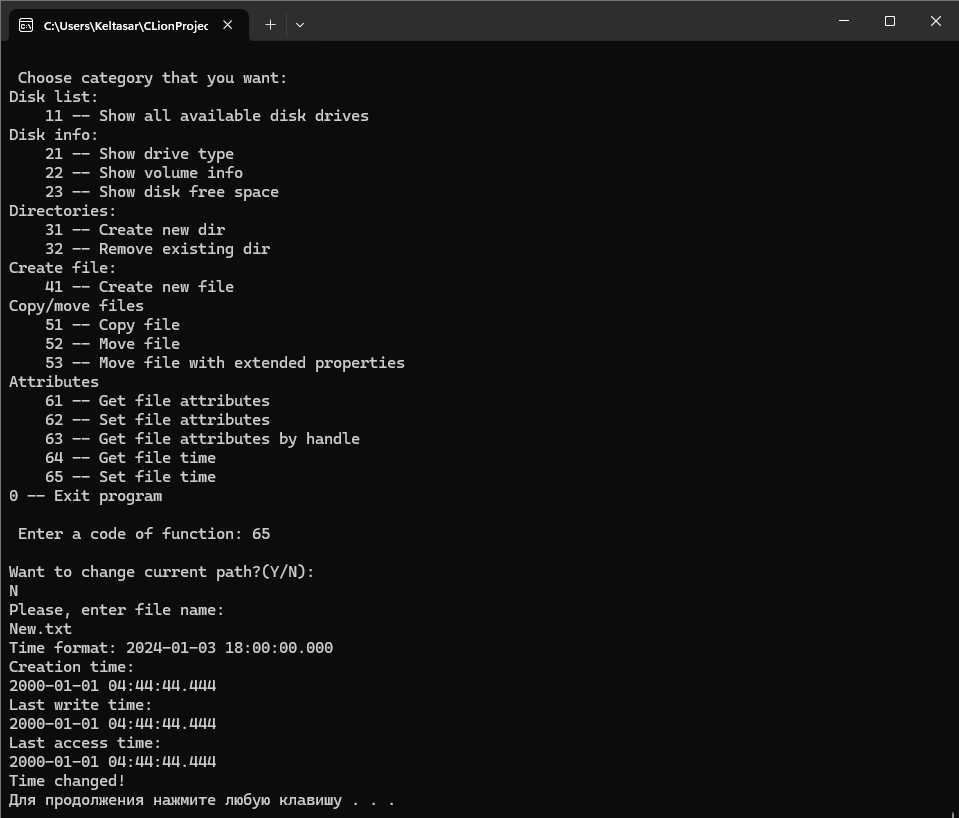


Рисунок 23: Изменение значений времени файла

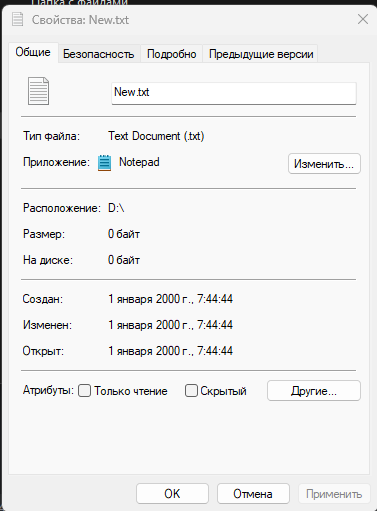


Рисунок 24: Значения времени файла в свойствах

## 2.7. Исходный код программы

#include <iostream>  
#include <windows.h>  
#include <string>  
#include <cmath>  
  
using namespace std;  
  
string currentPath = "D:\\";  
  
void MainMenu();  
void GetDList();  
void GetDType();  
void GetVInfo();  
void GetDSpaceInfo();  
void CreateRemoveDir(char action);  
void CreateNewFile();  
void FileCopyMove(char action);  
void FileAttributesGet();  
void FileAttributesSet();  
void GetFileInformationByHandle();  
void GetFileTime();  
void SetFileTime();  
string GetDName();  
void ChangeCurrentPath();  
  
int main(int argc, char\* argv[]) {  
 MainMenu();  
  
 return 0;  
}  
  
void MainMenu(){  
 int flag = -1;  
 do{  
 cout << "\n Choose category that you want: \n";  
 cout << "Disk list: \n";  
 cout << " 11 -- Show all available disk drives \n";  
 cout << "Disk info: \n";  
 cout << " 21 -- Show drive type \n";  
 cout << " 22 -- Show volume info \n";  
 cout << " 23 -- Show disk free space \n";  
 cout << "Directories: \n";  
 cout << " 31 -- Create new dir \n";  
 cout << " 32 -- Remove existing dir \n";  
 cout << "Create file: \n";  
 cout << " 41 -- Create new file \n";  
 cout << "Copy/move files \n";  
 cout << " 51 -- Copy file \n";  
 cout << " 52 -- Move file \n";  
 cout << " 53 -- Move file with extended properties \n";  
 cout << "Attributes \n";  
 cout << " 61 -- Get file attributes \n";  
 cout << " 62 -- Set file attributes \n";  
 cout << " 63 -- Get file attributes by handle \n";  
 cout << " 64 -- Get file time \n";  
 cout << " 65 -- Set file time \n";  
 cout << "0 -- Exit program \n";  
 cout << "\n Enter a code of function: ";  
  
 cin >> flag;  
 cout << "\n";  
  
  
 switch (flag){  
 case 0:  
 cout << "Bye! \n";  
 break;  
 case 1:  
 cout << "Current path: " << currentPath << "\n";  
 break;  
 case 11:  
 GetDList();  
 break;  
 case 21:  
 GetDType();  
 break;  
 case 22:  
 GetVInfo();  
 break;  
 case 23:  
 GetDSpaceInfo();  
 break;  
 case 31:  
 CreateRemoveDir('c');  
 break;  
 case 32:  
 CreateRemoveDir('r');  
 break;  
 case 41:  
 CreateNewFile();  
 break;  
 case 51:  
 FileCopyMove('c');  
 break;  
 case 52:  
 FileCopyMove('m');  
 break;  
 case 53:  
 FileCopyMove('e');  
 break;  
 case 61:  
 FileAttributesGet();  
 break;  
 case 62:  
 FileAttributesSet();  
 break;  
 case 63:  
 GetFileInformationByHandle();  
 break;  
 case 64:  
 GetFileTime();  
 break;  
 case 65:  
 SetFileTime();  
 break;  
 default:  
 cout << "Incorrect input, try again \n";  
 break;  
 }  
  
 system("pause");  
 }  
 while (flag != 0);  
}  
// ----- Disk List -----  
void GetDList(){  
 int dNumber = 1;  
 int dDetection;  
 char dLetter[4] = {0, ':', '\\', 0};  
 DWORD dr = GetLogicalDrives();  
 for (int i = 0; i < 26; i++){  
 dDetection = ((dr >> i) & 0x00000001);  
  
 if (dDetection == 1){  
 dLetter[0] = char(65 + i);  
  
 cout << dNumber << ". " << dLetter << "\n";  
 dNumber++;  
 }  
 }  
}  
// ----- Disk Info -----  
void GetDType(){  
 int d;  
 string n;  
  
 n = GetDName();  
  
 // <string variable>.c\_str() means that you convert to <const char \* type>, because "" isn't <const char \*> type  
  
 d = GetDriveType(n.c\_str());  
 cout << n << " is";  
 //cout << n + " is";  
  
 if (d == DRIVE\_UNKNOWN)  
 {  
 cout << " UNKNOWN" << endl;  
 }  
 if (d == DRIVE\_NO\_ROOT\_DIR)  
 {  
 cout << " DRIVE NO ROOT DIR" << endl;  
 }  
 if (d == DRIVE\_REMOVABLE)  
 {  
 cout << " REMOVABLE" << endl;  
 }  
 if (d == DRIVE\_FIXED)  
 {  
 cout << " FIXED" << endl;  
 }  
 if (d == DRIVE\_REMOTE)  
 {  
 cout << " REMOTE" << endl;  
 }  
 if (d == DRIVE\_CDROM)  
 {  
 cout << " CDROM" << endl;  
 }  
 if (d == DRIVE\_RAMDISK)  
 {  
 cout << " RAMDISK" << endl;  
 }  
}  
  
void GetVInfo(){  
 char volumeNameBuffer[100];  
 char fileSystemNameBuffer[100];  
 string n;  
 unsigned long volumeSerialNumber;  
  
 n = GetDName();  
  
 BOOL GetVolumeInformationFlag = GetVolumeInformationA(  
 n.c\_str(),  
 volumeNameBuffer,  
 100,  
 &volumeSerialNumber,  
 nullptr, //&MaximumComponentLength,  
 nullptr, //&FileSystemFlags,  
 fileSystemNameBuffer,  
 100  
 );  
  
 if (GetVolumeInformationFlag != 0)  
 if (GetVolumeInformationFlag != 0)  
{  
 cout << " Volume Name is " << volumeNameBuffer << endl;//When volume name is empty, that means, that your volume is unnamed  
 cout << " Volume Serial Number is " << volumeSerialNumber << endl;  
 cout << " File System is " << fileSystemNameBuffer << endl;  
 cout << " File System Flags: " << endl;  
 if (flags & FILE\_CASE\_SENSITIVE\_SEARCH)  
 cout << "- Case sensitive search" << endl;  
 if (flags & FILE\_CASE\_PRESERVED\_NAMES)  
 cout << "- Preserved case names" << endl;  
 if (flags & FILE\_UNICODE\_ON\_DISK)  
 cout << "- Unicode on disk" << endl;  
 if (flags & FILE\_PERSISTENT\_ACLS)  
 cout << "- Persistent ACLs" << endl;  
 if (flags & FILE\_FILE\_COMPRESSION)  
 cout << "- File compression" << endl;  
 if (flags & FILE\_VOLUME\_QUOTAS)  
 cout << "- Volume quotas" << endl;  
}

else  
 {  
 cout << " Not Present (GetVolumeInformation)" << endl;  
 }  
}  
  
void GetDSpaceInfo(){  
 string dName = GetDName();  
  
 //sec - sector, clus - cluster  
 DWORD secPerClus;  
 DWORD bytePerSec;  
 DWORD freeClus;  
 DWORD totalClus;  
 int dFreeSpace = GetDiskFreeSpace(dName.c\_str(), &secPerClus, &bytePerSec, &freeClus, &totalClus);  
  
 if (dFreeSpace != 0)  
 {  
 cout << "Disk " << dName << " free space info:\n";  
 cout << "Number of Sectors Per Cluster: " << (unsigned long long)secPerClus << "\nTotal Number Of Bytes per Sector: " << (unsigned long long)bytePerSec << "\nTotal Number Of Free Clusters: " << (unsigned long long)freeClus << "\nTotal Number Of Clusters: " << (unsigned long long)totalClus << "\nReturned value: " << (unsigned long long)dFreeSpace << "\n";  
 cout << "Total Number Of Bytes per Cluster: " << (unsigned long long)secPerClus\*(unsigned long long)bytePerSec << "\nTotal Number Of Free Sectors: " << (unsigned long long)freeClus\*(unsigned long long)secPerClus;  
 cout << "\nTotal Number Of Sectors: " << (unsigned long long)totalClus\*(unsigned long long)secPerClus << "\nTotal Number Of Free Bytes: " << (unsigned long long)freeClus\*(unsigned long long)secPerClus\*(unsigned long long)bytePerSec << "\nTotal Number of Bytes: " << (unsigned long long)totalClus\*(unsigned long long)secPerClus\*(unsigned long long)bytePerSec << "\n";  
 cout << "Total Number Of Used Clusters: " << (unsigned long long)totalClus - (unsigned long long)freeClus;  
 cout << "\nTotal Number Of Used Sectors: " << (unsigned long long)totalClus\*(unsigned long long)secPerClus - (unsigned long long)freeClus\*(unsigned long long)secPerClus << "\nTotal Number Of Used Bytes: " << (unsigned long long)totalClus\*(unsigned long long)secPerClus\*(unsigned long long)bytePerSec - (unsigned long long)freeClus\*(unsigned long long)secPerClus\*(unsigned long long)bytePerSec << "\n";  
 }  
 else  
 {  
 cout << "Returned value: " << (unsigned long long)dFreeSpace << "\n There is no such disk as " << dName << " :(\n";  
 }  
}  
  
// -----Directories-----  
void CreateRemoveDir(char action){  
 char t;  
 cout << "Want to change current path?(Y/N):\n";  
 cin >> t;  
  
 if (t == 'Y'){  
 ChangeCurrentPath();  
 }  
  
 string dirName;  
 cout << "Please, enter directory name:\n";  
 cin >> dirName;  
  
 std::string fullPath = currentPath + "\\" + dirName;  
  
 if (action == 'c'){  
 if (CreateDirectory(fullPath.c\_str(), NULL)){  
 cout << "Directory was successfully created!\n";  
 }  
 else{  
 cout << "Error with creating a directory:(\n";  
 }  
 }  
 else{  
 if (RemoveDirectory(fullPath.c\_str())){  
 cout << "Directory was successfully removed\n";  
 }  
 else{  
 cout << "Error with removing directory:(\n";  
 }  
 }  
}  
// ----- Create File -----  
void CreateNewFile(){  
 char t;  
 cout << "Want to change current path?(Y/N):\n";  
 cin >> t;  
  
 if (t == 'Y'){  
 ChangeCurrentPath();  
 }  
  
 string fileName;  
 cout << "Please, enter file name:\n";  
 cin >> fileName;  
  
 std::string fullPath = currentPath + "\\" + fileName;  
 HANDLE hFile = CreateFile(fullPath.c\_str(), GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL);  
  
 if (hFile == INVALID\_HANDLE\_VALUE) {  
 std::cerr << "Error with creating file!\n" << std::endl;  
 } else {  
 std::cout << "File succesfully created:\n" << fullPath << std::endl;  
 CloseHandle(hFile);  
 }  
}  
// ----- Copy/move file -----  
void FileCopyMove(char action){  
 char t;  
 cout << "Want to change current path?(Y/N):\n";  
 cin >> t;  
  
 if (t == 'Y'){  
 ChangeCurrentPath();  
 }  
  
 string fileName, copiedFileName;  
 cout << "Please, enter file name:\n";  
 cin >> fileName;  
 string sourcePath = currentPath + "\\" + fileName;  
 if (action == 'c') {  
 cout << "Enter copied file name:\n";  
 cin >> copiedFileName;  
 string destinationPath = currentPath + "\\" + copiedFileName;  
  
 if (CopyFileA(sourcePath.c\_str(), destinationPath.c\_str(), FALSE)) {  
 cout << "File successfully copied!\n";  
 } else {  
 cout << "Error while copying file:(\n";  
 }  
 }  
 if (action == 'm'){  
 string destinationPath;  
 cout << "Enter destination path:\n";  
 cin >> destinationPath;  
 destinationPath += "\\" + fileName;  
  
 if (MoveFile(sourcePath.c\_str(), destinationPath.c\_str())){  
 cout << "File successfully moved:\n";  
 }  
 else{  
 cout << "Error while moving file. Error code:" << GetLastError() << endl;  
 }  
 }  
 if (action == 'e'){  
 string destinationPath;  
 cout << "Enter destination path:\n";  
 destinationPath += "\\" + fileName;  
  
 DWORD flag = 2;  
 cout << "Please, choose the possible flags for moving file:\n"  
 << "1 -- MOVEFILE\_REPLACE\_EXISTING (replacing existing, it it exists)\n"  
 << "2 -- MOVEFILE\_COPY\_ALLOWED (classic move) -- DEFAULT\n"  
 << "4 -- MOVEFILE\_DELAY\_UNTIL\_REBOOT (moving after rebooting the system)\n"  
 << "8 -- MOVEFILE\_WRITE\_THROUGH (moving file, than returning value)\n"  
 << "16 -- MOVEFILE\_CREATE\_HARDLINK\n"  
 << "32 -- MOVEFILE\_FAIL\_IF\_NOT\_TRACKABLE\n";  
 cin >> flag;  
  
 if(MoveFileEx(currentPath.c\_str(),destinationPath.c\_str(),(DWORD)flag)){  
 cout << "File successfully moved with extensions!\n";  
 }  
 else{  
 cout << "Error while moving file. Error code:" << GetLastError() << endl;  
 }  
 }  
}  
// ----- Attributes -----  
void FileAttributesGet(){  
 char t;  
 cout << "Want to change current path?(Y/N):\n";  
 cin >> t;  
  
 if (t == 'Y'){  
 ChangeCurrentPath();  
 }  
  
 string fileName, copiedFileName;  
 cout << "Please, enter file name:\n";  
 cin >> fileName;  
 string filePath = currentPath + "\\" + fileName;  
  
 DWORD fileAttributes = 0;  
 fileAttributes = GetFileAttributes(filePath.c\_str());  
  
 cout << "File \"" << filePath << "\" attributes:\n";  
 if (fileAttributes & FILE\_ATTRIBUTE\_ARCHIVE)  
 {  
 cout << "Archive (FILE\_ATTRIBUTE\_ARCHIVE)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_COMPRESSED)  
 {  
 cout << "Compressed (FILE\_ATTRIBUTE\_COMPRESSED)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_DEVICE)  
 {  
 cout << "Device (FILE\_ATTRIBUTE\_DEVICE)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_DIRECTORY)  
 {  
 cout << "Directory (FILE\_ATTRIBUTE\_DIRECTORY)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_ENCRYPTED)  
 {  
 cout << "Encrypted (FILE\_ATTRIBUTE\_ENCRYPTED)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_HIDDEN)  
 {  
 cout << "Hidden (FILE\_ATTRIBUTE\_HIDDEN)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_INTEGRITY\_STREAM)  
 {  
 cout << "Data stream configured with integrity (FILE\_ATTRIBUTE\_INTEGRITY\_STREAM)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_NORMAL)  
 {  
 cout << "Normal (FILE\_ATTRIBUTE\_NORMAL)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED)  
 {  
 cout << "Not indexed (FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_NO\_SCRUB\_DATA)  
 {  
 cout << "Data stream not to be read by the data integrity scanner (FILE\_ATTRIBUTE\_NO\_SCRUB\_DATA)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_OFFLINE)  
 {  
 cout << "Don't avaliable immediatly (FILE\_ATTRIBUTE\_OFFLINE)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_READONLY)  
 {  
 cout << "Read-only (FILE\_ATTRIBUTE\_READONLY)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_RECALL\_ON\_DATA\_ACCESS)  
 {  
 cout << "Data is not fully presented locally (FILE\_ATTRIBUTE\_RECALL\_ON\_DATA\_ACCESS)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_RECALL\_ON\_OPEN)  
 {  
 cout << "Data hasn't physical representation on system (FILE\_ATTRIBUTE\_RECALL\_ON\_OPEN)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_REPARSE\_POINT)  
 {  
 cout << "Reparse point/representation link (FILE\_ATTRIBUTE\_REPARSE\_POINT)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_SPARSE\_FILE)  
 {  
 cout << "Sparse file (FILE\_ATTRIBUTE\_SPARSE\_FILE)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_SYSTEM)  
 {  
 cout << "System used (FILE\_ATTRIBUTE\_SYSTEM)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_TEMPORARY)  
 {  
 cout << "Temporary storage (FILE\_ATTRIBUTE\_TEMPORARY)\n";  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_VIRTUAL)  
 {  
 cout << "Reserved for system (FILE\_ATTRIBUTE\_VIRTUAL)\n";  
 }  
}  
void FileAttributesSet(){  
 char t;  
 cout << "Want to change current path?(Y/N):\n";  
 cin >> t;  
  
 if (t == 'Y'){  
 ChangeCurrentPath();  
 }  
  
 string fileName, copiedFileName;  
 cout << "Please, enter file name:\n";  
 cin >> fileName;  
 string filePath = currentPath + "\\" + fileName;  
  
 DWORD fileAttributes = 0;  
 fileAttributes = GetFileAttributes(filePath.c\_str());  
  
 DWORD attNum = 128;  
 cout << "Please, choose the possible attributes for the file/directory:\n"  
 << "1 -- FILE\_ATTRIBUTE\_READONLY (read-only)\n"  
 << "2 -- FILE\_ATTRIBUTE\_HIDDEN (hidden)\n"  
 << "4 -- FILE\_ATTRIBUTE\_SYSTEM (system used)\n"  
 << "32 -- FILE\_ATTRIBUTE\_ARCHIVE (archive)\n"  
 << "128 -- FILE\_ATTRIBUTE\_NORMAL -- DEFAULT\n"  
 << "256 -- FILE\_ATTRIBUTE\_TEMPORARY (temporary storage)\n"  
 << "4096 -- FILE\_ATTRIBUTE\_OFFLINE (don't avaliable immediatly)\n"  
 << "8192 -- FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED (not indexed)\n";  
 cin >> attNum;  
  
 if (SetFileAttributes(filePath.c\_str(), (DWORD) attNum))  
 {  
 cout << "The file's (directory's) \"" << filePath << "\" attributes has been successfully changed to:\n";  
 cout << "File \"" << filePath << "\" attributes:\n";  
 if (attNum & FILE\_ATTRIBUTE\_ARCHIVE)  
 {  
 cout << "Archive (FILE\_ATTRIBUTE\_ARCHIVE)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_COMPRESSED)  
 {  
 cout << "Compressed (FILE\_ATTRIBUTE\_COMPRESSED)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_DEVICE)  
 {  
 cout << "Device (FILE\_ATTRIBUTE\_DEVICE)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_DIRECTORY)  
 {  
 cout << "Directory (FILE\_ATTRIBUTE\_DIRECTORY)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_ENCRYPTED)  
 {  
 cout << "Encrypted (FILE\_ATTRIBUTE\_ENCRYPTED)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_HIDDEN)  
 {  
 cout << "Hidden (FILE\_ATTRIBUTE\_HIDDEN)\n";  
 }  
 /\*if (inFunctionNumber & FILE\_ATTRIBUTE\_INTEGRITY\_STREAM)  
 {  
 cout << "Data stream configured with integrity (FILE\_ATTRIBUTE\_INTEGRITY\_STREAM)\n";  
 }\*/  
 if (attNum & FILE\_ATTRIBUTE\_NORMAL)  
 {  
 cout << "Normal (FILE\_ATTRIBUTE\_NORMAL)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED)  
 {  
 cout << "Not indexed (FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED)\n";  
 }  
 /\*if (inFunctionNumber & FILE\_ATTRIBUTE\_NO\_SCRUB\_DATA)  
 {  
 cout << "Data stream not to be read by the data integrity scanner (FILE\_ATTRIBUTE\_NO\_SCRUB\_DATA)\n";  
 }\*/  
 if (attNum & FILE\_ATTRIBUTE\_OFFLINE)  
 {  
 cout << "Don't avaliable immediatly (FILE\_ATTRIBUTE\_OFFLINE)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_READONLY)  
 {  
 cout << "Read-only (FILE\_ATTRIBUTE\_READONLY)\n";  
 }  
 /\*if (inFunctionNumber & FILE\_ATTRIBUTE\_RECALL\_ON\_DATA\_ACCESS)  
 {  
 cout << "Data is not fully presented locally (FILE\_ATTRIBUTE\_RECALL\_ON\_DATA\_ACCESS)\n";  
 }  
 if (inFunctionNumber & FILE\_ATTRIBUTE\_RECALL\_ON\_OPEN)  
 {  
 cout << "Data hasn't physical representation on system (FILE\_ATTRIBUTE\_RECALL\_ON\_OPEN)\n";  
 }\*/  
 if (attNum & FILE\_ATTRIBUTE\_REPARSE\_POINT)  
 {  
 cout << "Reparse point/representation link (FILE\_ATTRIBUTE\_REPARSE\_POINT)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_SPARSE\_FILE)  
 {  
 cout << "Sparse file (FILE\_ATTRIBUTE\_SPARSE\_FILE)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_SYSTEM)  
 {  
 cout << "System used (FILE\_ATTRIBUTE\_SYSTEM)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_TEMPORARY)  
 {  
 cout << "Temporary storage (FILE\_ATTRIBUTE\_TEMPORARY)\n";  
 }  
 if (attNum & FILE\_ATTRIBUTE\_VIRTUAL)  
 {  
 cout << "Reserved for system (FILE\_ATTRIBUTE\_VIRTUAL)\n";  
 }  
 }  
 else  
 {  
 cout << "Something wrong! The file's (directory's) \"" << filePath << "\" attributes hasn't been changed\n";  
 cout << "Last error code is \"" << GetLastError() << "\"\n"; // here i need to insert last error text string  
 }  
}  
  
void GetFileInformationByHandle ()  
{  
 char t;  
 cout << "Want to change current path?(Y/N):\n";  
 cin >> t;  
  
 if (t == 'Y'){  
 ChangeCurrentPath();  
 }  
  
 string fileName, copiedFileName;  
 cout << "Please, enter file name:\n";  
 cin >> fileName;  
 string filePath = currentPath + "\\" + fileName;  
  
 HANDLE hFile = CreateFile(filePath.c\_str(), // file name  
 GENERIC\_READ, // open for reading  
 0, // do not share  
 NULL, // default security  
 OPEN\_EXISTING, // existing file only  
 FILE\_ATTRIBUTE\_NORMAL, // normal file  
 NULL);  
 int size=0;  
 //PBY\_HANDLE\_FILE\_INFORMATION lpFileInformation = new \_BY\_HANDLE\_FILE\_INFORMATION();  
 BY\_HANDLE\_FILE\_INFORMATION\* lpFileInformation = new BY\_HANDLE\_FILE\_INFORMATION();  
 int resalt = GetFileInformationByHandle(hFile,lpFileInformation);  
 size = lpFileInformation->nFileSizeLow;  
  
 DWORD localAttributes = lpFileInformation->dwFileAttributes;  
  
 if (localAttributes & FILE\_ATTRIBUTE\_ARCHIVE)  
 {  
 cout << "Archive (FILE\_ATTRIBUTE\_ARCHIVE)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_COMPRESSED)  
 {  
 cout << "Compressed (FILE\_ATTRIBUTE\_COMPRESSED)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_DEVICE)  
 {  
 cout << "Device (FILE\_ATTRIBUTE\_DEVICE)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_DIRECTORY)  
 {  
 cout << "Directory (FILE\_ATTRIBUTE\_DIRECTORY)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_ENCRYPTED)  
 {  
 cout << "Encrypted (FILE\_ATTRIBUTE\_ENCRYPTED)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_HIDDEN)  
 {  
 cout << "Hidden (FILE\_ATTRIBUTE\_HIDDEN)\n";  
 }

if (localAttributes & FILE\_ATTRIBUTE\_NORMAL)  
 {  
 cout << "Normal (FILE\_ATTRIBUTE\_NORMAL)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED)  
 {  
 cout << "Not indexed (FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED)\n";  
 }

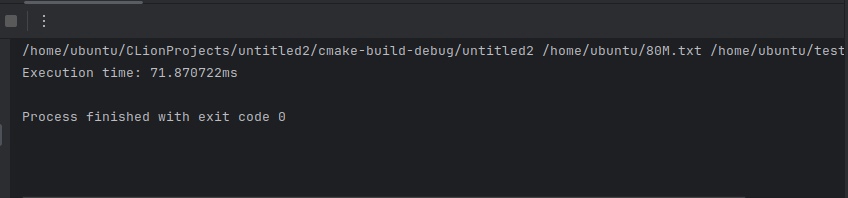
if (localAttributes & FILE\_ATTRIBUTE\_OFFLINE)  
 {  
 cout << "Don't avaliable immediatly (FILE\_ATTRIBUTE\_OFFLINE)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_READONLY)  
 {  
 cout << "Read-only (FILE\_ATTRIBUTE\_READONLY)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_REPARSE\_POINT)  
 {  
 cout << "Reparse point/representation link (FILE\_ATTRIBUTE\_REPARSE\_POINT)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_SPARSE\_FILE)  
 {  
 cout << "Sparse file (FILE\_ATTRIBUTE\_SPARSE\_FILE)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_SYSTEM)  
 {  
 cout << "System used (FILE\_ATTRIBUTE\_SYSTEM)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_TEMPORARY)  
 {  
 cout << "Temporary storage (FILE\_ATTRIBUTE\_TEMPORARY)\n";  
 }  
 if (localAttributes & FILE\_ATTRIBUTE\_VIRTUAL)  
 {  
 cout << "Reserved for system (FILE\_ATTRIBUTE\_VIRTUAL)\n";  
 }  
  
 char buffer[256];  
  
 SYSTEMTIME st;  
 FILETIME ft;  
 string strMessage;  
  
 // first  
  
 ft.dwLowDateTime = (lpFileInformation->ftCreationTime).dwLowDateTime;  
 ft.dwHighDateTime = (lpFileInformation->ftCreationTime).dwHighDateTime;  
  
 FileTimeToSystemTime(&ft, &st);  
  
 sprintf( buffer,  
 "%d-%02d-%02d %02d:%02d:%02d.%03d",  
 st.wYear,  
 st.wMonth,  
 st.wDay,  
 st.wHour,  
 st.wMinute,  
 st.wSecond,  
 st.wMilliseconds );  
 strMessage = buffer;  
  
 std::cout << "CREATION TIME = " << strMessage << endl;  
  
 // second  
  
 ft.dwLowDateTime = (lpFileInformation->ftLastWriteTime).dwLowDateTime;  
 ft.dwHighDateTime = (lpFileInformation->ftLastWriteTime).dwHighDateTime;  
  
 FileTimeToSystemTime(&ft, &st);  
  
 sprintf( buffer,  
 "%d-%02d-%02d %02d:%02d:%02d.%03d",  
 st.wYear,  
 st.wMonth,  
 st.wDay,  
 st.wHour,  
 st.wMinute,  
 st.wSecond,  
 st.wMilliseconds );  
 strMessage = buffer;  
  
 std::cout << "LAST WRITE TIME = " << strMessage << endl;  
  
 // third  
  
 ft.dwLowDateTime = (lpFileInformation->ftLastAccessTime).dwLowDateTime;  
 ft.dwHighDateTime = (lpFileInformation->ftLastAccessTime).dwHighDateTime;  
  
 FileTimeToSystemTime(&ft, &st);  
  
 sprintf( buffer,  
 "%d-%02d-%02d %02d:%02d:%02d.%03d",  
 st.wYear,  
 st.wMonth,  
 st.wDay,  
 st.wHour,  
 st.wMinute,  
 st.wSecond,  
 st.wMilliseconds );  
 strMessage = buffer;  
  
 std::cout << "LAST ACCESS TIME = " << strMessage << endl;  
  
 cout << "Volume serial number: " << lpFileInformation->dwVolumeSerialNumber << "\n";  
 cout << "Local size high/low: " << lpFileInformation->nFileSizeHigh << " " << lpFileInformation->nFileSizeLow << "\n";  
 cout << "Number Of Links: " << lpFileInformation->nNumberOfLinks << "\n";  
 cout << "Index High/low: " << lpFileInformation->nFileIndexHigh << " " << lpFileInformation->nFileIndexLow << "\n";  
  
 CloseHandle(hFile);  
}  
  
void GetFileTime ()  
{  
 char t;  
 cout << "Want to change current path?(Y/N):\n";  
 cin >> t;  
  
 if (t == 'Y'){  
 ChangeCurrentPath();  
 }  
  
 string fileName, copiedFileName;  
 cout << "Please, enter file name:\n";  
 cin >> fileName;  
 string filePath = currentPath + "\\" + fileName;  
  
 HANDLE hFile = CreateFile(filePath.c\_str(), // file name  
 GENERIC\_READ, // open for reading  
 0, // do not share  
 NULL, // default security  
 OPEN\_EXISTING, // existing file only  
 0, // normal file  
 NULL);  
  
 FILETIME creationTime;  
 FILETIME lastWriteTime;  
 FILETIME lastAccessTime;  
  
 if (GetFileTime(hFile, &creationTime, &lastAccessTime, &lastWriteTime)){  
 char buffer[256];  
  
 SYSTEMTIME st;  
 FileTimeToSystemTime(&creationTime, &st);  
  
 sprintf( buffer,  
 "%d-%02d-%02d %02d:%02d:%02d.%03d",  
 st.wYear,  
 st.wMonth,  
 st.wDay,  
 st.wHour,  
 st.wMinute,  
 st.wSecond,  
 st.wMilliseconds );  
 string strMessage = buffer;  
  
 std::cout << "System time = " << strMessage << std::endl;  
  
 FileTimeToSystemTime(&lastAccessTime, &st);  
  
 sprintf( buffer,  
 "%d-%02d-%02d %02d:%02d:%02d.%03d",  
 st.wYear,  
 st.wMonth,  
 st.wDay,  
 st.wHour,  
 st.wMinute,  
 st.wSecond,  
 st.wMilliseconds );  
 strMessage = buffer;  
  
 std::cout << "Access time = " << strMessage << std::endl;  
  
 FileTimeToSystemTime(&lastWriteTime, &st);  
  
 sprintf( buffer,  
 "%d-%02d-%02d %02d:%02d:%02d.%03d",  
 st.wYear,  
 st.wMonth,  
 st.wDay,  
 st.wHour,  
 st.wMinute,  
 st.wSecond,  
 st.wMilliseconds );  
 strMessage = buffer;  
  
 std::cout << "Change time = " << strMessage << std::endl;  
 }  
 else  
 {  
 cout << "Something wrong!" << "\n";  
 }  
 CloseHandle(hFile);  
}

void SetFileTime ()  
{  
 char t;  
 cout << "Want to change current path?(Y/N):\n";  
 cin >> t;  
  
 if (t == 'Y'){  
 ChangeCurrentPath();  
 }  
  
 string fileName, copiedFileName;  
 cout << "Please, enter file name:\n";  
 cin >> fileName;  
 string filePath = currentPath + "\\" + fileName;  
  
 HANDLE hFile = CreateFile(filePath.c\_str(), // file name  
 GENERIC\_WRITE, // open for reading  
 0, // do not share  
 NULL, // default security  
 OPEN\_EXISTING, // existing file only  
 0, // normal file  
 NULL);  
  
 string one;  
 string two;  
 string three;  
  
 cout << "Time format: 2024-01-03 18:00:00.000\n";  
 cout << "Creation time:\n";  
 fflush(stdin);  
 getline(cin, one);  
  
 cout << "Last write time:\n";  
 fflush(stdin);  
 getline(cin, two);  
  
  
 cout << "Last access time:\n";  
 fflush(stdin);  
 getline(cin, three);  
  
 FILETIME creationTime;//= buffTime.creationTime;  
 FILETIME lastWriteTime;// = buffTime.lastWriteTime;  
 FILETIME lastAccessTime;// = buffTime.lastAccessTime;  
  
 SYSTEMTIME systime\_1;  
 SYSTEMTIME systime\_2;  
 SYSTEMTIME systime\_3;  
  
 memset(&systime\_1,0,sizeof(systime\_1));  
 // Date string should be "yyyy-MM-dd hh:mm"  
 sscanf\_s(one.c\_str(), "%d-%d-%d%d:%d:%d:%d.%d",  
 &systime\_1.wYear,  
 &systime\_1.wMonth,  
 &systime\_1.wDay,  
 &systime\_1.wHour,  
 &systime\_1.wMinute,  
 &systime\_1.wSecond,  
 &systime\_1.wMilliseconds);  
  
 SystemTimeToFileTime(&systime\_1, &creationTime);  
  
 memset(&systime\_2,0,sizeof(systime\_2));  
 // Date string should be "yyyy-MM-dd hh:mm"  
 sscanf\_s(two.c\_str(), "%d-%d-%d%d:%d:%d:%d.%d",  
 &systime\_2.wYear,  
 &systime\_2.wMonth,  
 &systime\_2.wDay,  
 &systime\_2.wHour,  
 &systime\_2.wMinute,  
 &systime\_2.wSecond,  
 &systime\_2.wMilliseconds);  
  
 SystemTimeToFileTime(&systime\_2, &lastWriteTime);  
  
 memset(&systime\_3,0,sizeof(systime\_3));  
 // Date string should be "yyyy-MM-dd hh:mm"  
 sscanf\_s(three.c\_str(), "%d-%d-%d%d:%d:%d:%d.%d",  
 &systime\_3.wYear,  
 &systime\_3.wMonth,  
 &systime\_3.wDay,  
 &systime\_3.wHour,  
 &systime\_3.wMinute,  
 &systime\_3.wSecond,  
 &systime\_3.wMilliseconds);  
  
 SystemTimeToFileTime(&systime\_3, &lastAccessTime);  
  
 if(SetFileTime(hFile, &creationTime, &lastAccessTime, &lastWriteTime))  
 {  
 cout << "Time changed!" << endl;  
 }  
 else  
 {  
 cout << "Time hasn't been changed!" << endl;  
 }  
  
 CloseHandle(hFile);  
}  
// ----- Additional Functions -----  
string GetDName ()  
{  
 string d;  
  
 cout << "Please, input disk name you want:\n";  
 cin >> d;  
 d = d + ":\\";  
  
 return d;  
}  
  
void ChangeCurrentPath(){  
 cout << "Please, enter new directory path (example \"C:\\DirName1\\DirName2):\n";  
 cin >> currentPath;  
 SetCurrentDirectory(currentPath.c\_str());  
}

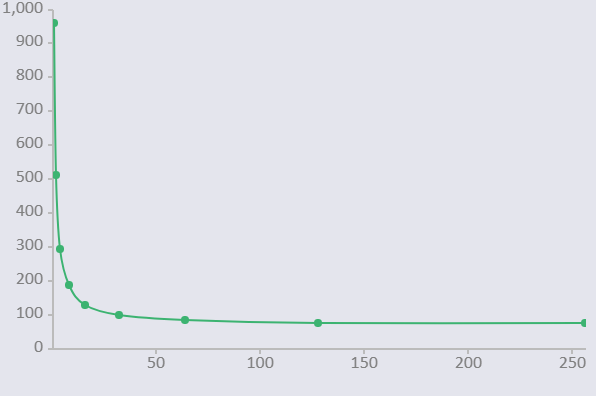
## 2.8. Выводы

В ходе выполнения первой части лабораторной работы были изучены основные функции управления файловой системой. С помощью функций Win32 API было создано приложение, которое реализовывало управление дисками, каталогами и файлами. Во-первых, вывод списков дисков и получение необходимой информации о них. Во-вторых, создание и удаление выбранной директории. В-третьих, работа с файлами: их копирование, перемещение, создание, получение их данных и атрибутов (и изменение). Таким образом и было реализовано управление дисками, файлами и каталогами.

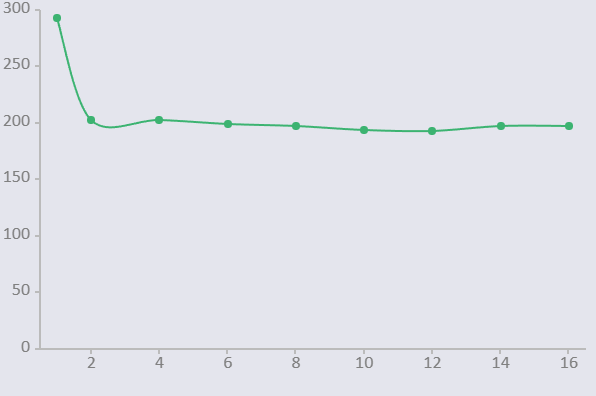
1. **Копирование файла с помощью перекрывающих операций ввода/вывода**
   1. **Демонстрация работы программы**



*Рисунок 25. Копирование текстового файла размером 80 МБ при помощи 16 операций и блоками размером 16\*4096 байт*

**

*Рисунок 26. Зависимость времени копирования в мс от размера блока(х\*4096 байт) при 1 операции ввода-вывода*

**

*Рисунок 27 зависимость времени копирования в мс от кол-ва операций при размере блока в 4\*4096 байт*

Можно заметить что время сильно зависит от размера блока, но не может быть ниже определенного порога около 200 мс. При этом если отнять это время то зависимость будет приблизительно обратной. Это связяно с тем что программе необходимо выполнить ряд действий кроме чтения записи. (открытие файла, инициализация операций и т.д.). От количества операций зависимость времени не такая значительная. Наименьшее значение времени выполнения достигается при 12 операциях. Вероятно конкретное значение зависит от многих параметров и может изменяться, также судя по всему количество свободных потоков является «бутылочным горлышком» при копировании. Все тесты проводились на файле размера 80 Мб.

* 1. **Исходный код**

#include <chrono>

#include <iomanip>

#include <cstdio>

#include <cstdlib>

#include <aio.h>

#include <fcntl.h>

#include <unistd.h>

#include <csignal>

#include <cstring>

#include <cerrno>

#include <sys/types.h>

#include <sys/stat.h>

#include <csignal>

#include <cstdint>

#include <cinttypes>

#include <iostream>

struct aio\_operation {

struct aiocb controlBlock;

char \*buffer;

int isWriteOperation;

bool done;

};

int fileDescriptorFrom;

int fileDescriptorTo;

unsigned long long fileSize;

unsigned blockSize;

aio\_operation\* operations;

int operationsN = 0;

int workingOperations = 0;

bool initFinished = false;

void aio\_completion\_handler(sigval\_t sigval) {

auto \*aio\_op = (struct aio\_operation \*)sigval.sival\_ptr;

if (aio\_op->done) return;

if (aio\_op->controlBlock.aio\_offset >= fileSize) {

delete aio\_op->buffer;

aio\_op->done = true;

workingOperations--;

} else if (aio\_op->isWriteOperation) {

aio\_op->controlBlock.aio\_fildes = fileDescriptorTo;

if (aio\_op->controlBlock.aio\_offset + aio\_op->controlBlock.aio\_nbytes > fileSize) {

aio\_op->controlBlock.aio\_nbytes = fileSize - aio\_op->controlBlock.aio\_offset;

}

aio\_write(&aio\_op->controlBlock);

} else {

aio\_op->controlBlock.aio\_offset += aio\_op->controlBlock.aio\_nbytes \* operationsN; {

aio\_op->controlBlock.aio\_fildes = fileDescriptorFrom;

aio\_read(&aio\_op->controlBlock);

}

}

aio\_op->isWriteOperation = !aio\_op->isWriteOperation;

}

void initCopy(std::string pathFrom, std::string pathTo, int operationsNumber = 1, int blockSizeMultiplayer = 1 /\*in clusters\*/) {

fileDescriptorFrom = open(pathFrom.c\_str(), O\_RDONLY | O\_NONBLOCK, 0666);

fileDescriptorTo = open(pathTo.c\_str(), O\_CREAT | O\_WRONLY | O\_TRUNC | O\_NONBLOCK, 0666);

struct stat statBuffer{};

fstat(fileDescriptorFrom, &statBuffer);

fileSize = statBuffer.st\_size;

blockSize = statBuffer.st\_blksize \* blockSizeMultiplayer;

operations = new aio\_operation[operationsNumber];

operationsN = operationsNumber;

for (int i = 0; i < operationsNumber; i++) {

operations[i].isWriteOperation = 1;

operations[i].buffer = new char[blockSize]{0};

operations[i].controlBlock = aiocb{0};

operations[i].controlBlock.aio\_fildes = fileDescriptorFrom;

operations[i].controlBlock.aio\_buf = operations[i].buffer;

operations[i].controlBlock.aio\_nbytes = blockSize;

operations[i].controlBlock.aio\_offset = i \* blockSize;

operations[i].controlBlock.aio\_sigevent.sigev\_notify = SIGEV\_THREAD;

operations[i].controlBlock.aio\_sigevent.sigev\_notify\_function = aio\_completion\_handler;

operations[i].controlBlock.aio\_sigevent.sigev\_value.sival\_ptr = &operations[i];

operations[i].done = false;

workingOperations++;

// initialisation of aio operation

aio\_read(&operations[i].controlBlock);

}

initFinished = true;

}

int main(int argc, char \*\*argv) { // args are "from" filepath, "to" filepath, block size in cluster sizes, operations number

if (argc != 5) {

std::cerr << "Wrong number of arguments. Must be 4:" << std::endl;

return 1;

}

const std::string oldFilePath(argv[1]); // old file

const std::string newFilePath(argv[2]); // new file

// const unsigned long oldFileBytes = atoi(argv[3]); // size

int blockSizeMultiplayer = atoi(argv[3]); // block size in clusters

int operations = atoi(argv[4]); // operations number

auto start = std::chrono::high\_resolution\_clock::now();

initCopy(oldFilePath, newFilePath, operations, blockSizeMultiplayer);

while (workingOperations) { sleep(0); } // todo

auto end = std::chrono::high\_resolution\_clock::now();

auto elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start).count() / 1e6;

std::cout << "Execution time: " << std::setprecision(10) << elapsed << "ms" << std::endl;

return 0;

}

* 1. **Выводы**

В ходе выполнения 2-й части лабораторной работы была реализована программа, выполняющая копирование файла при помощи ассинхронного ввода-вывода.Код демонстрирует использование асинхронного ввода-вывода для чтения и записи файлов. Это позволяет выполнять операции ввода-вывода параллельно, что может значительно увеличить производительность программы. В результате выполнения лабораторной работы был создан код, который копирует содержимое одного файла в другой асинхронно.

Влияние характеристик системы на копирование при помощи асинхронного ввода-вывода может быть значительным. Например, количество операций ввода-вывода, которые могут быть выполнены одновременно, может зависеть от количества доступных ядер процессора и объема оперативной памяти. Также скорость чтения и записи файлов может зависеть от скорости диска и его типа (например, SSD или HDD). Кроме того, скорость копирования может быть ограничена скоростью сети, если файлы находятся на удаленном сервере. В целом, для достижения наилучшей производительности при копировании файлов с использованием асинхронного ввода-вывода, рекомендуется использовать систему с достаточным количеством ресурсов и быстрыми устройствами хранения данных. Поскольку все тесты делались через ОС Ubuntu, установленную на внешний HDD(за отсутствием альтернатив), что сильно влияло на связь с ЦП и ОЗУ, из-за чего скорость копирования оказалась гораздо меньше, чем могла быть на установленном непосредственно стоящем в системе HDD.