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. 9308 |  | Соболев М.С. |
| Преподаватель |  | Тимофеев А.В. |

Оглавление

[1. Введение 3](#_Toc84214820)

[2. Управление дисками, каталогами и файлами 6](#_Toc84214821)

[2.1. Вывод списка дисков 6](#_Toc84214822)

[2.2. Вывод информации о диске и размер свободного пространства 7](#_Toc84214823)

[2.3. Создание и удаление заданных каталогов 10](#_Toc84214824)

[2.4. Создание файлов в новых каталогах 12](#_Toc84214825)

[2.5. Копирование и перемещение файлов между каталогами 15](#_Toc84214826)

[2.6. Анализ и изменение атрибутов файлов 18](#_Toc84214827)

[2.7. Исходный код программы 24](#_Toc84214828)

[2.8. Выводы 80](#_Toc84214829)

[3. Копирование файла с помощью перекрывающих операций ввода-вывода 81](#_Toc84214830)

[3.1. Создание и запуск консольного приложения 81](#_Toc84214831)

[3.2. Проверка приложения на разных размерах копируемых блоков 82](#_Toc84214832)

[3.3. Проверка приложения на разном числе операция ввода-вывода 84](#_Toc84214833)

[3.4. Исходный код программы 85](#_Toc84214834)

[3.5. Выводы 94](#_Toc84214835)

[4. Список использованных источников 95](#_Toc84214836)

# 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 функций GetLogicalDrives() и GetLogicalDriveStrings();

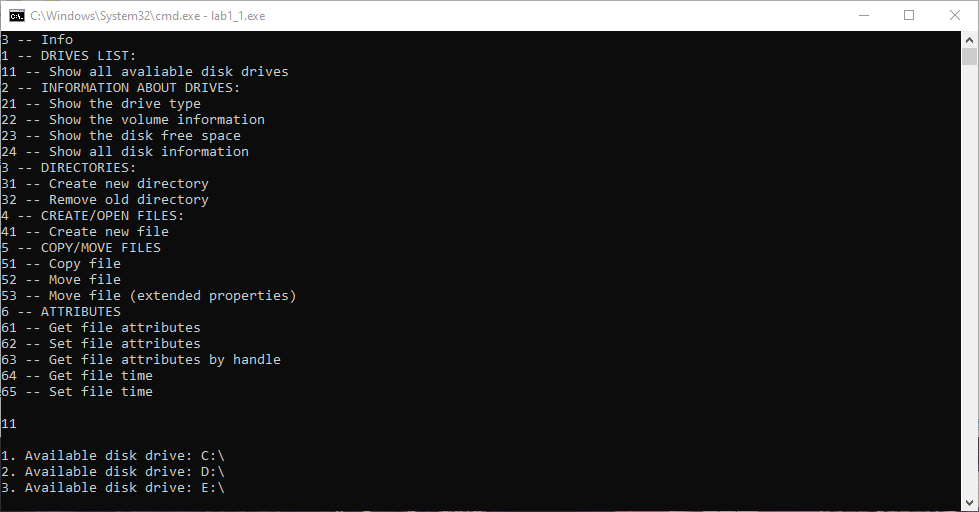


Рисунок 1: Вывод списка дисков

## 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().

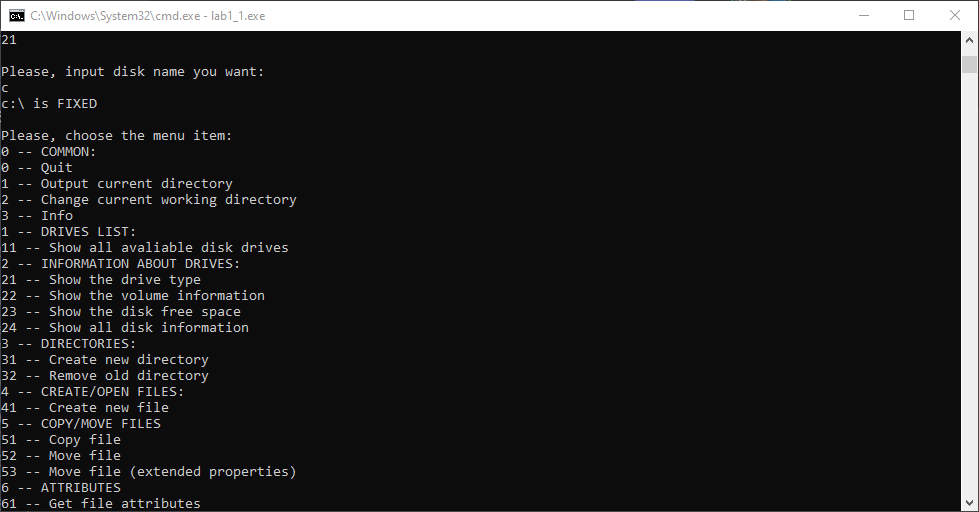


Рисунок 2: Вывод типа диска

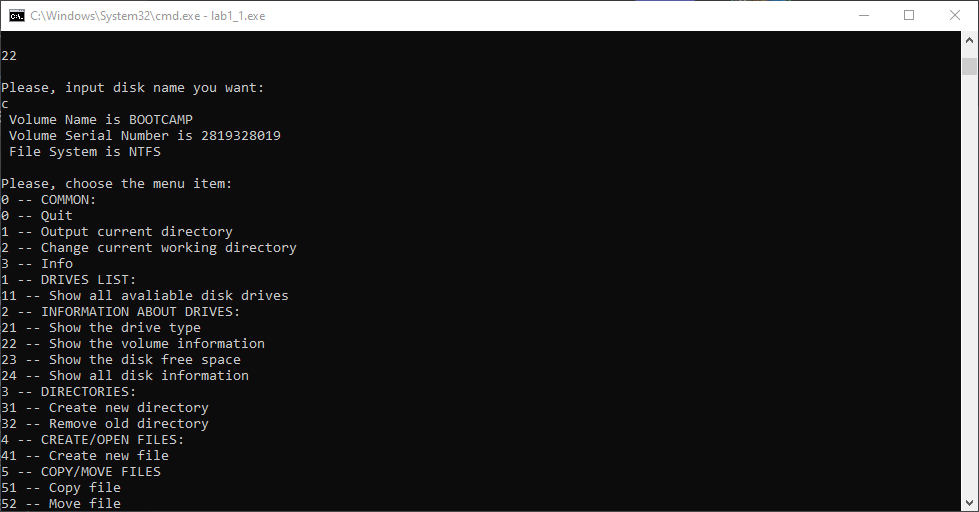


Рисунок 3: Вывод информации о диске

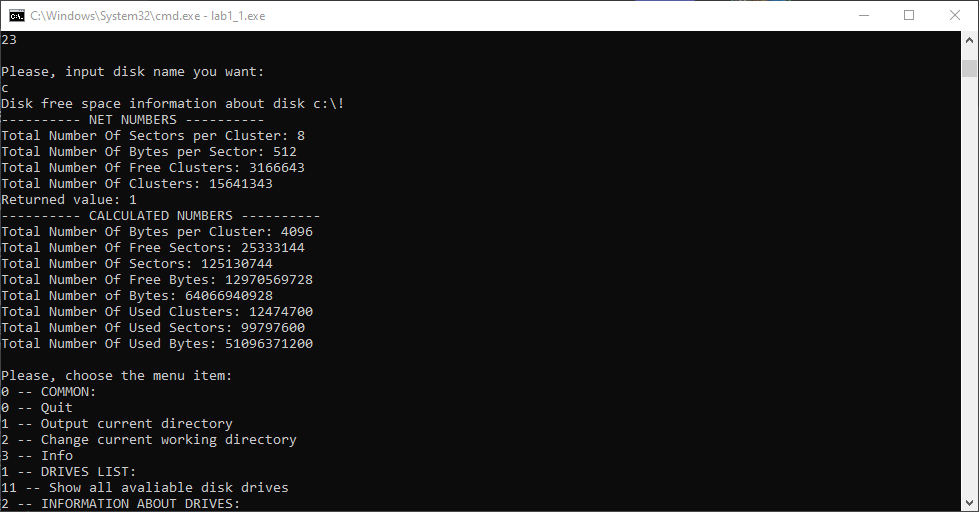


Рисунок 4: Вывод информации о пространстве на диске

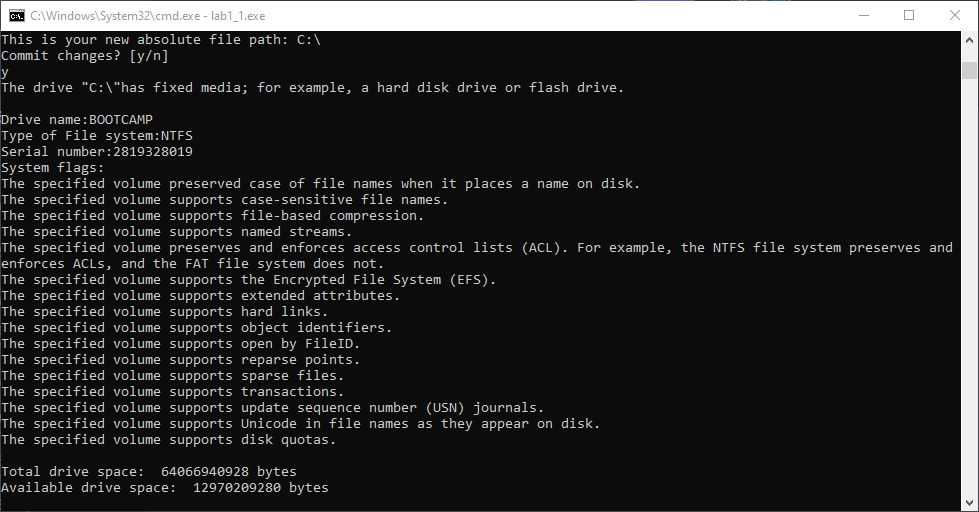


Рисунок 5: Вывод всей информации о диске

## 2.3. Создание и удаление заданных каталогов

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

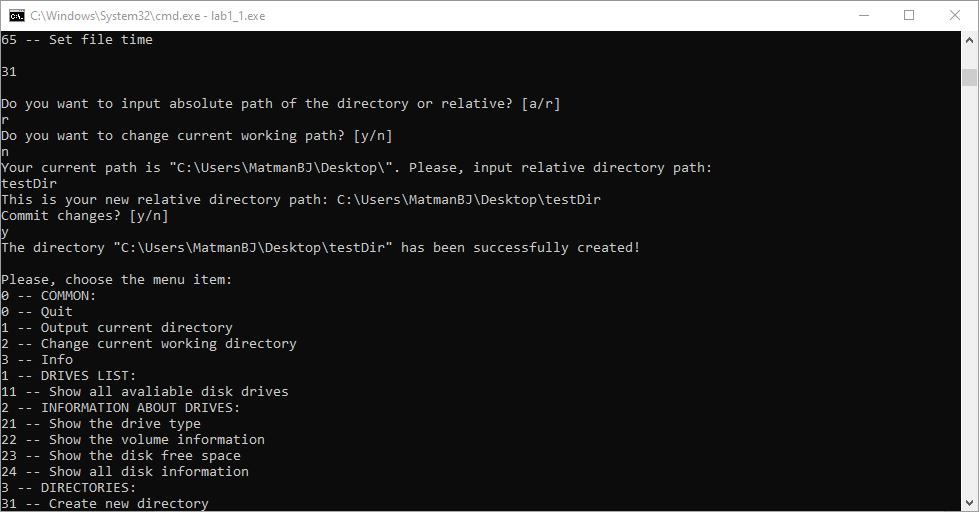


Рисунок 6: Создание каталога

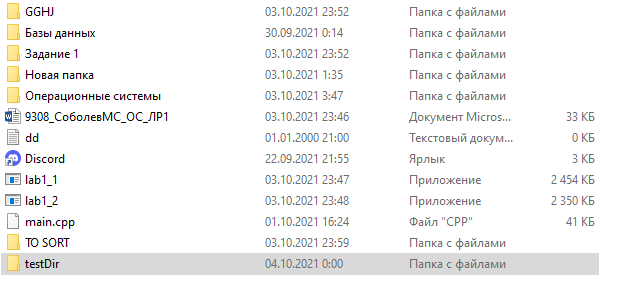


Рисунок 7: Директория каталога

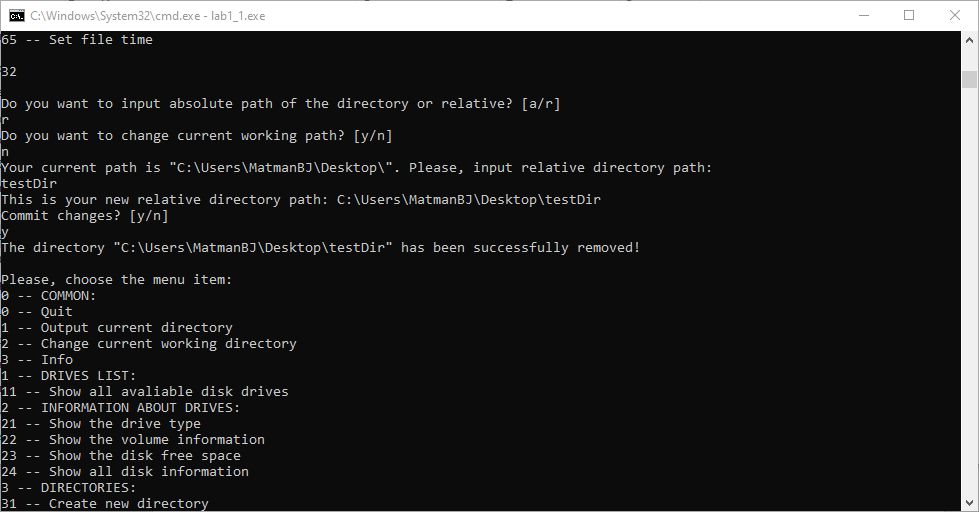


Рисунок 8: Удаление каталога

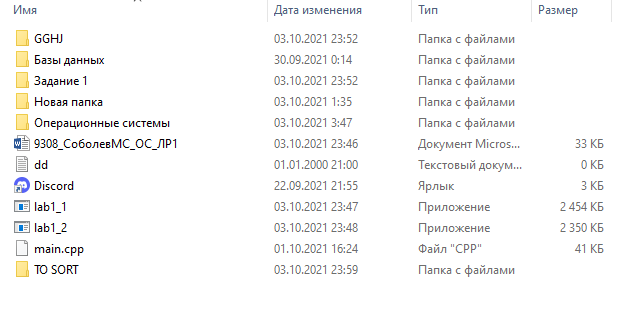


Рисунок 9: Директория каталога

## 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().

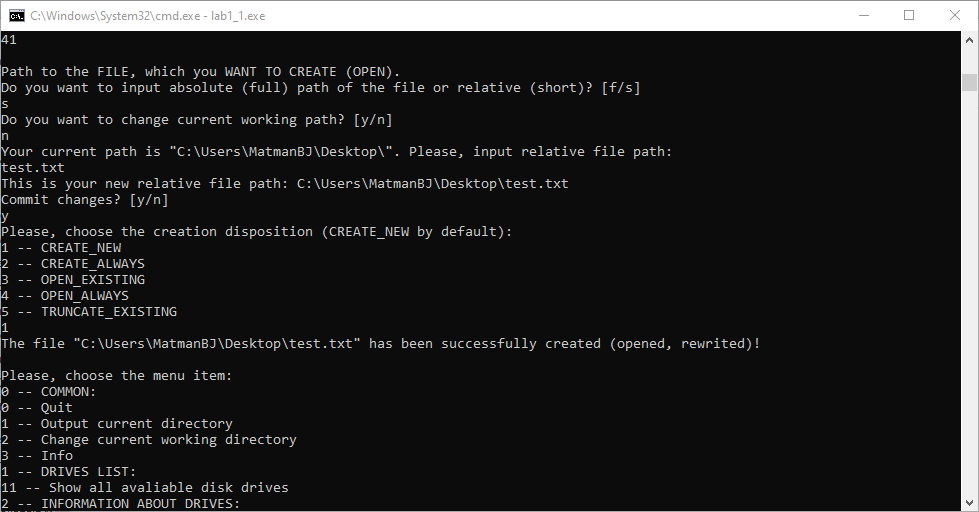


Рисунок 10: Создание файла

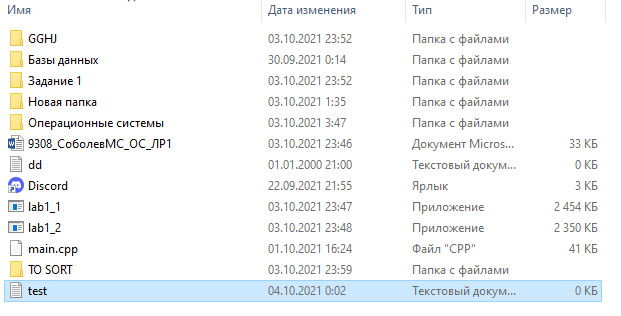


Рисунок 11: Директория файла

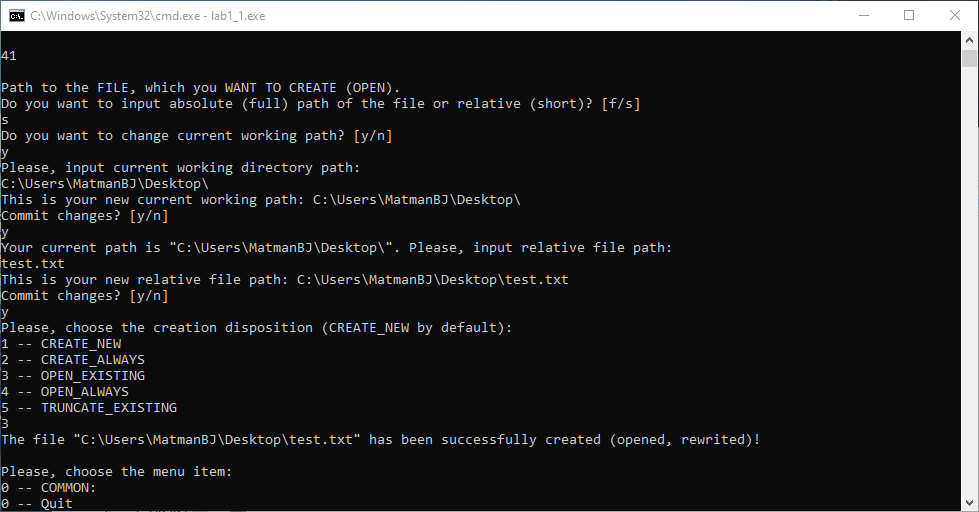


Рисунок 12: Открытие файла

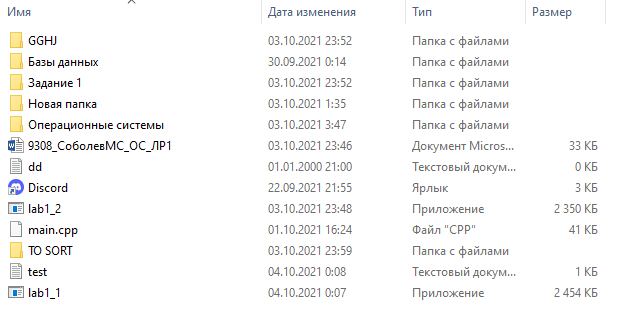


Рисунок 13: Директория файла

## 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.

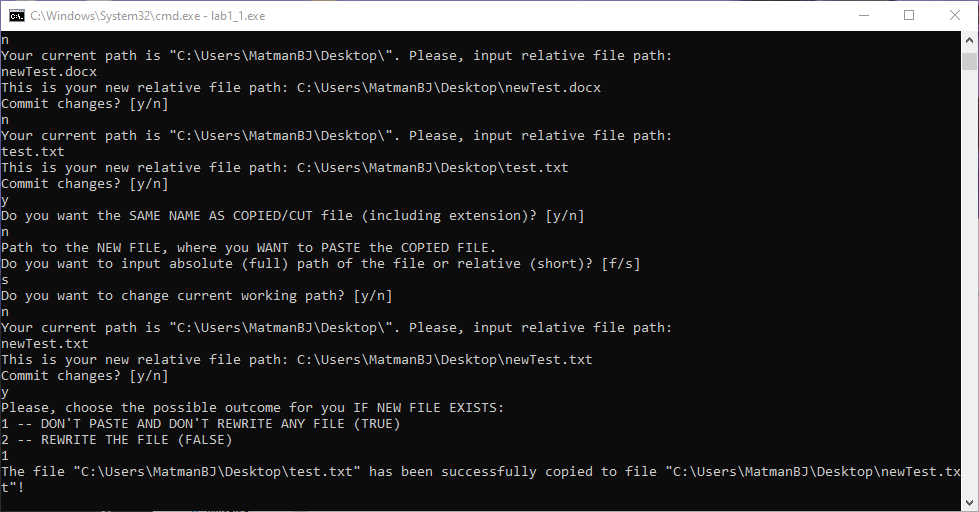


Рисунок 14: Копирование файла

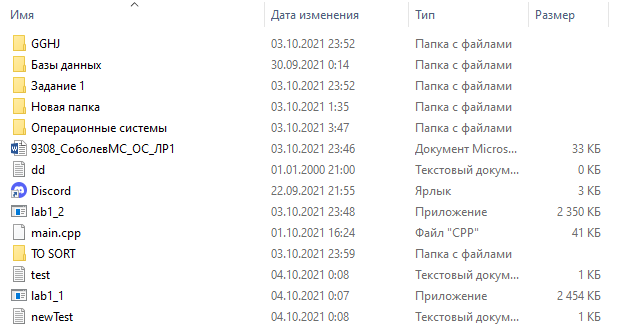


Рисунок 15: Директория файла

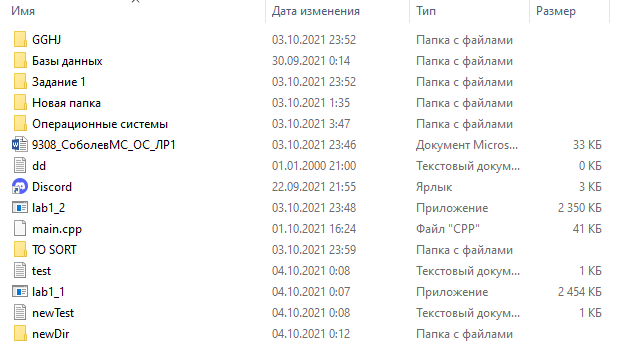


Рисунок 16: Директория файла до перемещения

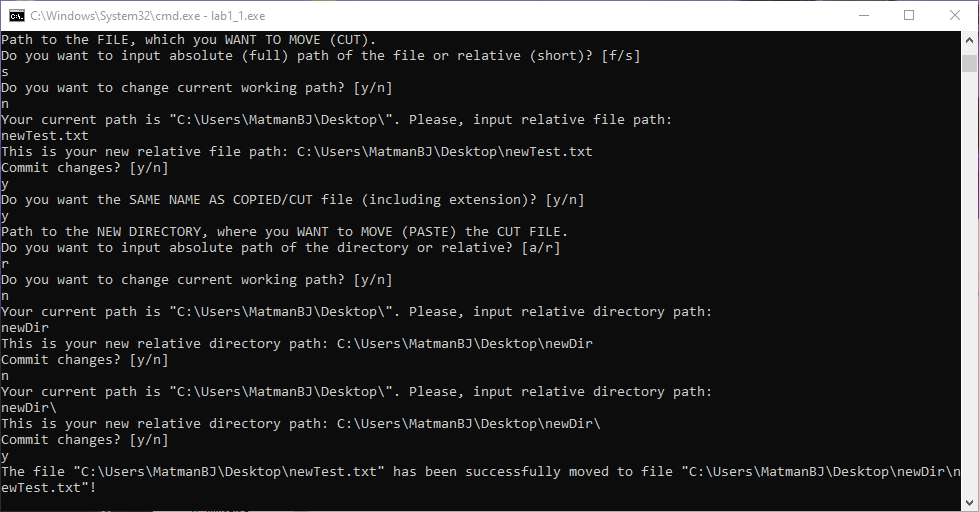


Рисунок 17: Перемещение файла

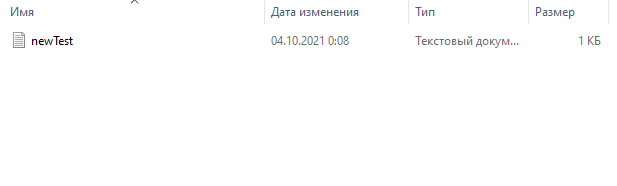


Рисунок 18: Директория файла после перемещения

## 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().

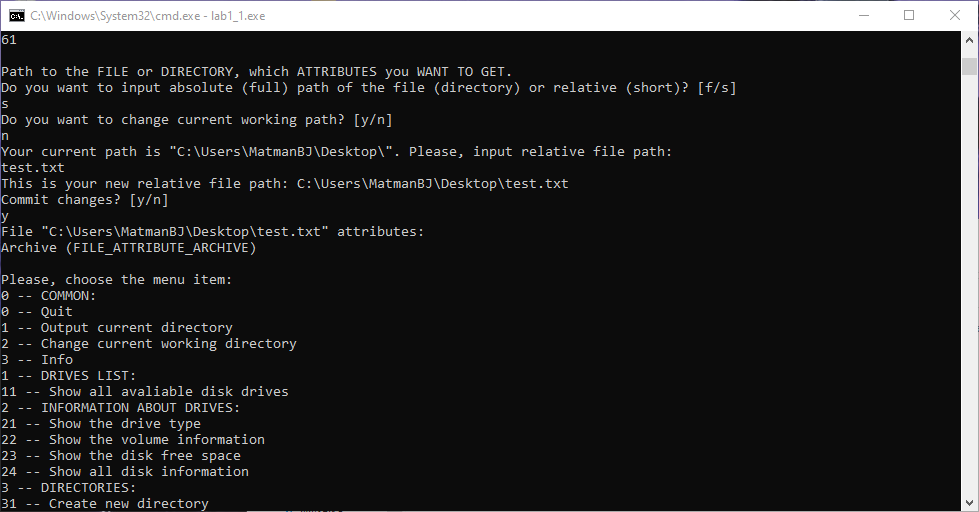


Рисунок 19: Вывод атрибутов файла

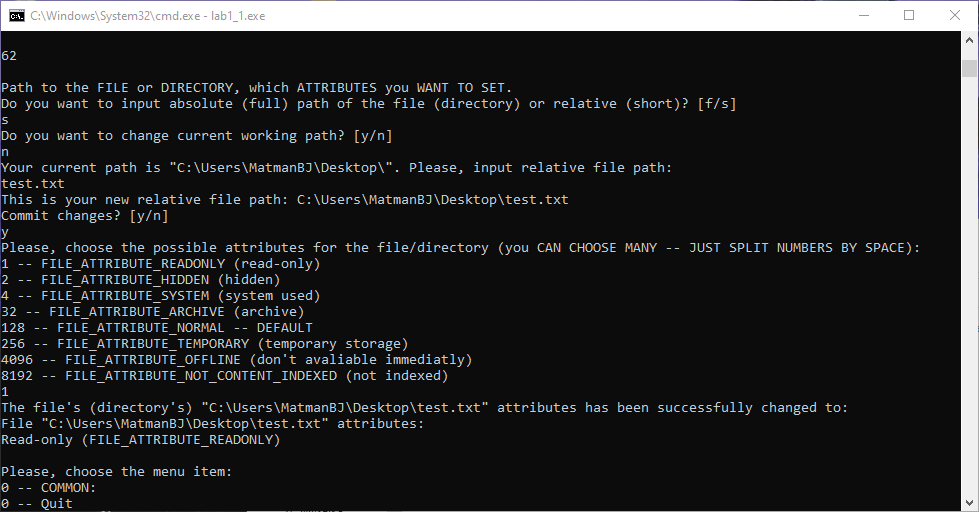


Рисунок 20: Изменение атрибутов файла

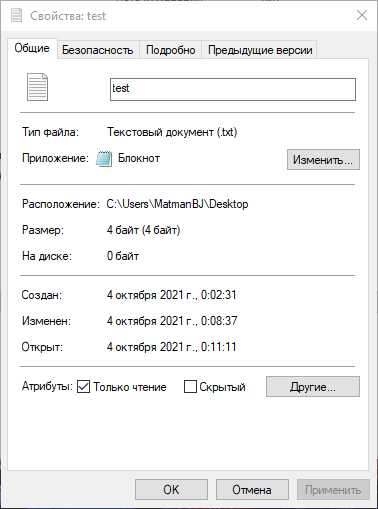


Рисунок 21: Изменённый атрибут файла в свойствах

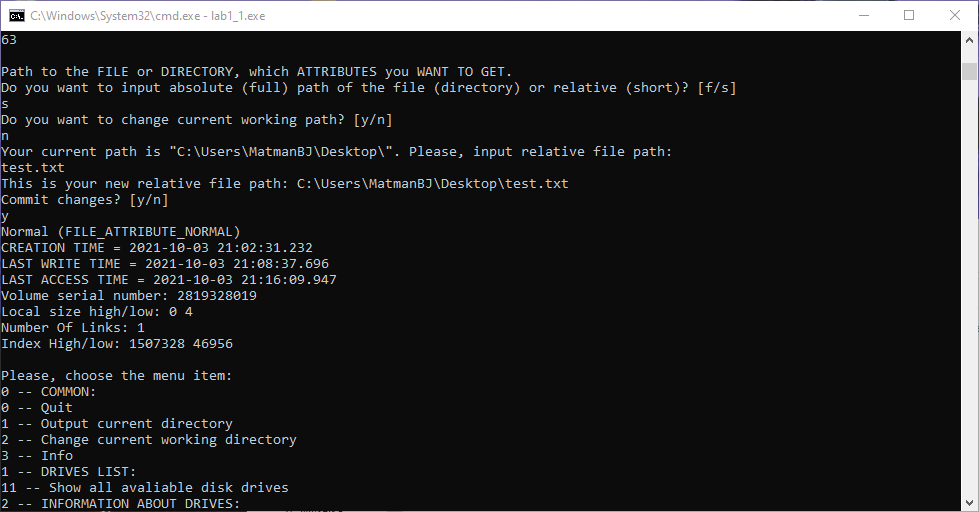


Рисунок 22: Вывод атрибутов файла по дескриптору

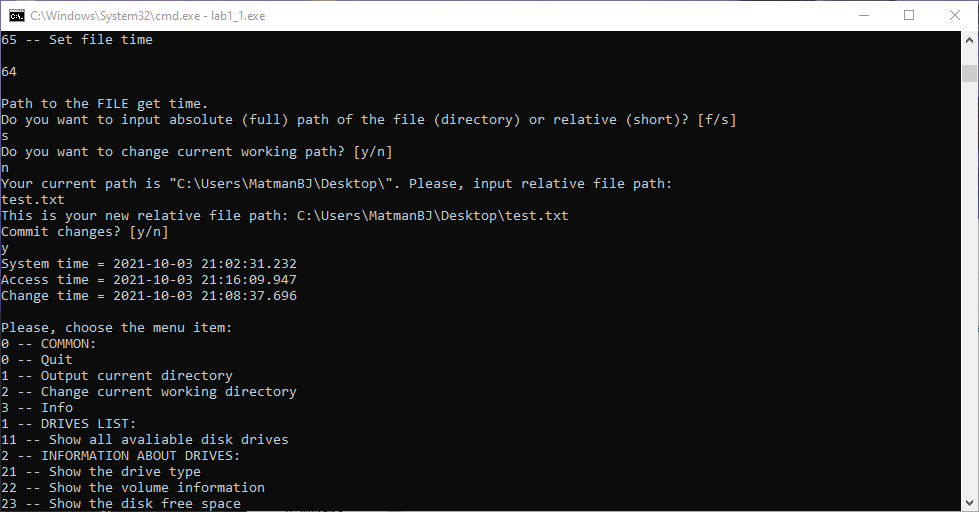


Рисунок 23: Вывод значений времени файла

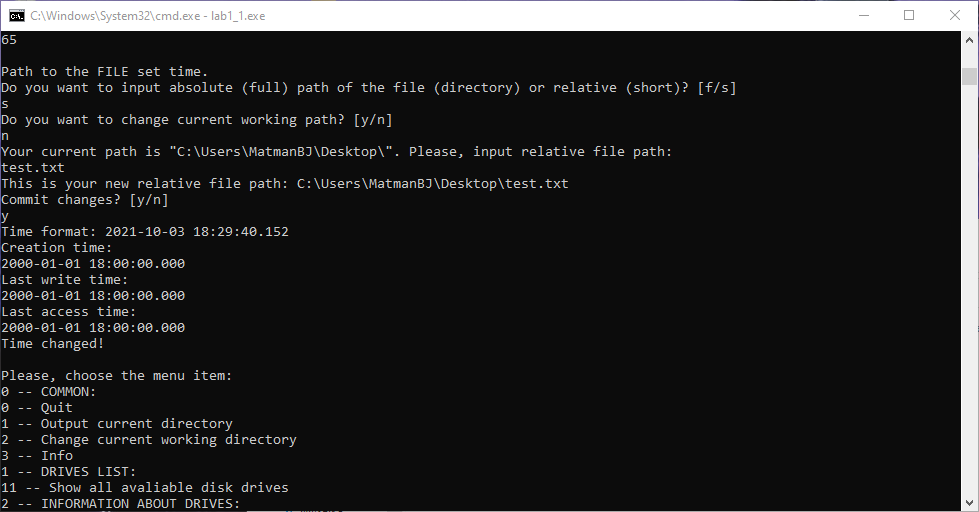


Рисунок 24: Изменение значений времени файла

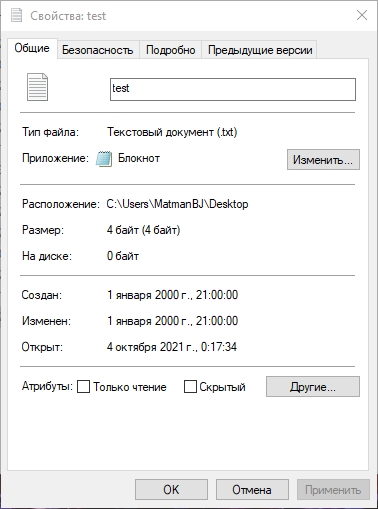


Рисунок 25: Значения времени файла в свойствах

## 2.7. Исходный код программы

/\*

Saint Petersburg Electrotechnical University "LETI" (ETU "LETI"),

Faculty of Computer Science and Technology "FKTI",

Department of Computer Science and Engineering,

Computer Systems Engineering and Informatics (09.03.01) program.

OS labortory work 1 version 0\_8 dated 2021\_09\_23

This software is under MIT License (X11 License).

You can see a detailed description in "LICENSE.md" file.

Copyight (c) 2021 Sobolev Matvey Sergeevich

\*/

/\*

Win32 API (WinAPI) is a set of functions in the library <windows.h>

API means "Application Programming Interface"

\*/

#include <windows.h> // for WinAPI functions

#include <math.h> // for double making

#include <exception> // for exceptions

#include <iostream> // just for working

#include <string> // for the "string" type using

#include <vector> // for the "vector" type using

#include <algorithm> // for the "find" function using

using namespace std;

string currentPath = "c:\\"; // the current working path is disc "c:/" by default!

// ---------- FUNCTION DECLARATION ----------

string GetDiskName();

void MainMenu ();

void Info ();

void LocalGetLogicalDrives ();

void LocalGetDriveType ();

void LocalGetVolumeInformation ();

void LocalGetDiskFreeSpaceEx ();

void LocalGetDiskFreeSpace ();

void GetDiskInfo();

bool dirExists(const std::string& dirName\_in);

string GetPathKernel (char localFlag);

string GetPathShell (char localFlagOne, char localFlagTwo, string localMessageOne, string localMessageTwo);

void LocalCreateRemoveDirectory (char actionCreateRemove);

void LocalCreateFile();

void LocalCopyMoveFile(char actionCopyMove);

void LocalGetFileAttributes ();

void LocalSetFileAttributes ();

void LocalGetFileInformationByHandle ();

void GetFileTime ();

void SetFileTime ();

// ---------- MAIN ----------

int main (int argc, char\* argv[]) // i've finally understood what it means (argc -- number of arguments, argv -- strings of arguments (including -<word> and --<word>))

{

// "GET CURRENT DIRECTORY", "SET CURRENT DIRECTORY"

int flag = -1; // "-1" for incorrect input continue the program

do

{

MainMenu();

cin >> flag;

cout << "\n";

switch (flag)

{

case 0:

cout << "Goodbye!";

break;

case 1:

cout << "Your current working path is: \"" << currentPath << "\" (c:\\ by default).\n";

break;

case 2:

currentPath = GetPathKernel('c');

break;

case 3:

Info();

break;

case 11:

LocalGetLogicalDrives();

break;

case 21:

LocalGetDriveType();

break;

case 22:

LocalGetVolumeInformation();

break;

case 23:

LocalGetDiskFreeSpace();

break;

case 24:

GetDiskInfo();

break;

case 231:

LocalGetDiskFreeSpaceEx();

break;

case 31:

LocalCreateRemoveDirectory ('c');

break;

case 32:

LocalCreateRemoveDirectory ('r');

break;

case 41:

LocalCreateFile();

break;

case 51:

LocalCopyMoveFile('c');

break;

case 52:

LocalCopyMoveFile('m');

break;

case 53:

LocalCopyMoveFile('e');

break;

case 61:

LocalGetFileAttributes();

break;

case 62:

LocalSetFileAttributes();

break;

case 63:

LocalGetFileInformationByHandle();

break;

case 64:

GetFileTime();

break;

case 65:

SetFileTime();

break;

default:

cout << "Incorrect input! Try again.";

break;

}

}

while (flag != 0);

return 0;

}

// ---------- 0 -- GET DISK NAME ----------

string GetDiskName ()

{

string localDisk;

cout << "Please, input disk name you want:\n";

cin >> localDisk;

localDisk = localDisk + ":\\";

return localDisk;

}

// ---------- 0 -- MAIN MENU ----------

void MainMenu ()

{

cout << "\n";

cout << "Please, choose the menu item:\n";

cout << "0 -- COMMON:\n";

cout << "0 -- Quit\n";

cout << "1 -- Output current directory\n";

cout << "2 -- Change current working directory\n";

cout << "3 -- Info\n";

cout << "1 -- DRIVES LIST:\n";

cout << "11 -- Show all avaliable disk drives\n";

cout << "2 -- INFORMATION ABOUT DRIVES:\n";

cout << "21 -- Show the drive type\n";

cout << "22 -- Show the volume information\n";

cout << "23 -- Show the disk free space\n";

cout << "24 -- Show all disk information\n";

cout << "3 -- DIRECTORIES:\n";

cout << "31 -- Create new directory\n";

cout << "32 -- Remove old directory\n";

cout << "4 -- CREATE/OPEN FILES:\n";

cout << "41 -- Create new file\n";

cout << "5 -- COPY/MOVE FILES\n";

cout << "51 -- Copy file\n";

cout << "52 -- Move file\n";

cout << "53 -- Move file (extended properties)\n";

cout << "6 -- 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 << "\n";

}

// ---------- 0 -- INFO ----------

void Info ()

{

cout << "Saint Petersburg Electrotechnical University \"LETI\" (ETU \"LETI\"),\n"

<< "Faculty of Computer Science and Technology \"FKTI\",\n"

<< "Department of Computer Science and Engineering,\n"

<< "Computer Systems Engineering and Informatics (09.03.01) program.\n\n"

<< "OS labortory work 1 version 0\_8 dated 2021\_09\_23\n\n"

<< "This software is under MIT License (X11 License).\n"

<< "You can see a detailed description in \"LICENSE.md\" file.\n\n"

<< "Copyight (c) 2021 Sobolev Matvey Sergeevich\n";

}

// ---------- 1 -- GET LOGICAL DRIVES ----------

void LocalGetLogicalDrives()

{

int localDriveNumber = 1;

int localDiskDetection;

char localDriveLetter[4];

DWORD dr = GetLogicalDrives();

for(int i = 0; i < 26; i++)

{

localDiskDetection = ((dr >> i) & 0x00000001);

if (localDiskDetection == 1)

{

localDriveLetter[0] = char(65 + i);

localDriveLetter[1] = ':';

localDriveLetter[2] = '\\';

localDriveLetter[3] = 0;

cout << localDriveNumber << ". Available disk drive: " << localDriveLetter << endl;

localDriveNumber = localDriveNumber + 1; // next number

}

}

}

// ---------- 2 -- GET DRIVE TYPE ----------

void LocalGetDriveType()

{

// i need to check, if i use the "uint" namespace

int d;

string n;

n = GetDiskName();

// <string variable>.c\_str() means that you convert to <const char \* type>, because "" isn't <const char \*> type

d = GetDriveType(n.c\_str()); // i just want to know what's going one if i choose d or f!

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;

}

}

// ---------- 2 -- GET VOLUME INFORMATION ----------

void LocalGetVolumeInformation ()

{

char VolumeNameBuffer[100];

char FileSystemNameBuffer[100];

string n;

unsigned long VolumeSerialNumber;

n = GetDiskName();

BOOL GetVolumeInformationFlag = GetVolumeInformationA(

n.c\_str(),

VolumeNameBuffer,

100,

&VolumeSerialNumber,

NULL, //&MaximumComponentLength,

NULL, //&FileSystemFlags,

FileSystemNameBuffer,

100

);

/\*BOOL GetVolumeInformationFlag = GetVolumeInformationA(

"d:\\",

VolumeNameBuffer,

100,

&VolumeSerialNumber,

NULL, //&MaximumComponentLength,

NULL, //&FileSystemFlags,

FileSystemNameBuffer,

100

);\*/

if (GetVolumeInformationFlag != 0)

{

cout << " Volume Name is " << VolumeNameBuffer << endl;

cout << " Volume Serial Number is " << VolumeSerialNumber << endl;

cout << " File System is " << FileSystemNameBuffer << endl;

}

else

{

cout << " Not Present (GetVolumeInformation)" << endl;

}

}

// ---------- 2 -- GET DISK FREE SPACE EX ----------

void LocalGetDiskFreeSpaceEx ()

{

DWORD FreeBytesAvailable;

DWORD TotalNumberOfBytes;

DWORD TotalNumberOfFreeBytes;

BOOL GetDiskFreeSpaceFlag = GetDiskFreeSpaceEx("c:\\", // directory name

(PULARGE\_INTEGER)&FreeBytesAvailable, // bytes available to caller

(PULARGE\_INTEGER)&TotalNumberOfBytes, // bytes on disk

(PULARGE\_INTEGER)&TotalNumberOfFreeBytes // free bytes on disk

);

if(GetDiskFreeSpaceFlag != 0)

{

//double d = double(unsigned long(TotalNumberOfFreeBytes))/1024/1024/1024;

//cout << d;

cout << " Total Number Of Free Bytes = " << (unsigned long)TotalNumberOfFreeBytes << "( " << (double)(((unsigned long)(TotalNumberOfFreeBytes/1024))/1000) << " Mb )" << endl;

cout << " Total Number Of Bytes = " << (unsigned long)TotalNumberOfBytes << "( " << (double)(((unsigned long)(TotalNumberOfBytes/1024))/1000) << " Mb )" << endl;

cout << " Total Number Of Bytes = " << (unsigned long)TotalNumberOfBytes << "( " << TotalNumberOfBytes << " Mb )" << endl;

unsigned long tmp\_1 = TotalNumberOfBytes;

tmp\_1 = tmp\_1/1024;

cout << tmp\_1;

double tmp\_2 = tmp\_1;

tmp\_2 = tmp\_2/1000;

cout << tmp\_2;

cout << TotalNumberOfBytes/1024000;

}

else

{

cout << " Not Present (GetDiskFreeSpace)" << endl;

}

}

// ---------- 2 -- GET DISK FREE SPACE ----------

void LocalGetDiskFreeSpace ()

{

/\*long unsigned int \* secPerClus;

long unsigned int \* bytePerSec;

long unsigned int \* freeClus;

long unsigned int \* totalClus;\*/

/\*LPDWORD secPerClus;

LPDWORD bytePerSec;

LPDWORD freeClus;

LPDWORD totalClus;\*/

string localDiskName = GetDiskName();

//const char diskNameCC[4] = {'e', ':', '\\'}; // you can do this

//string diskNameS = "e:\\"; // and you can do this

DWORD secPerClus;

DWORD bytePerSec;

DWORD freeClus;

DWORD totalClus;

//int gdfs = GetDiskFreeSpace(diskNameCC, &secPerClus, &bytePerSec, &freeClus, &totalClus); // const char\* explicitly

//int gdfs = GetDiskFreeSpace(diskNameS.c\_str(), &secPerClus, &bytePerSec, &freeClus, &totalClus); // const char\* from string (with c\_str() method)

int gdfs = GetDiskFreeSpace(localDiskName.c\_str(), &secPerClus, &bytePerSec, &freeClus, &totalClus);

if (gdfs != 0)

{

cout << "Disk free space information about disk " << localDiskName << "!\n";

cout << "---------- NET NUMBERS ----------\n";

cout << "Total 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)gdfs << "\n";

cout << "---------- CALCULATED NUMBERS ----------\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)gdfs << "\nThere is no such disk as " << localDiskName << "!\n";

}

}

// ---------- 2 -- GET DISK FREE SPACE ----------

void GetDiskInfo()

{

string drive = GetPathShell('f', 's', "Path to the DISK.\n", "Do you want to input absolute (full) path of the file (directory) or relative (short)? [f/s]\n");

DWORD drive\_type = GetDriveTypeA(drive.c\_str());

switch (drive\_type) {

case DRIVE\_UNKNOWN: cout<< "The drive type cannot be determined.\n";

break;

case DRIVE\_NO\_ROOT\_DIR: cout<< "The root path is invalid; for example, there is no volume mounted at the specified path. \n";

break;

case DRIVE\_REMOVABLE: cout<< "The drive \""<< drive<<"\" has removable media; for example, a floppy drive, thumb drive, or flash card reader. \n";

break;

case DRIVE\_FIXED: cout<< "The drive \""<< drive<<"\"has fixed media; for example, a hard disk drive or flash drive. \n";

break;

case DRIVE\_REMOTE: cout<< "The drive \""<< drive<<"\" is a remote (network) drive. \n";

break;

case DRIVE\_CDROM: cout<< "The drive \""<< drive<<"\" is a CD-ROM drive\n";

break;

case DRIVE\_RAMDISK: cout<< "The drive \""<< drive<<"\" is a RAM disk.\n";

break;

default: cout<< "You shouldn't see this massage. Smth goes wrong";

}

char nameBuffer[100];

char sysNameBuff[100];

DWORD serialNumber,maxComponentLength,fileSystemFlags;

if(GetVolumeInformationA(drive.c\_str(),nameBuffer,sizeof(nameBuffer),&serialNumber,&maxComponentLength,&fileSystemFlags,sysNameBuff,sizeof(sysNameBuff))){

cout << "\nDrive name:" << nameBuffer << endl <<

"Type of File system:" << sysNameBuff << endl <<

"Serial number:" << serialNumber << endl <<

"System flags:" << endl;

string specVol = "The specified volume";

string specVolSup = specVol + " supports";

if (fileSystemFlags & FILE\_CASE\_PRESERVED\_NAMES)

cout << specVol + " preserved case of file names when it places a name on disk.\n";

if (fileSystemFlags & FILE\_CASE\_SENSITIVE\_SEARCH)

cout << specVolSup + " case-sensitive file names.\n";

if (fileSystemFlags & FILE\_FILE\_COMPRESSION)

cout << specVolSup + " file-based compression.\n";

if (fileSystemFlags & FILE\_NAMED\_STREAMS)

cout << specVolSup + " named streams.\n";

if (fileSystemFlags & FILE\_PERSISTENT\_ACLS)

cout << specVol + " preserves and enforces access control lists (ACL). For example, the NTFS file system preserves and enforces ACLs, and the FAT file system does not.\n";

if (fileSystemFlags & FILE\_READ\_ONLY\_VOLUME)

cout << specVol + " is read-only.\n";

if (fileSystemFlags & FILE\_SEQUENTIAL\_WRITE\_ONCE)

cout << specVolSup + " a single sequential write.\n";

if (fileSystemFlags & FILE\_SUPPORTS\_ENCRYPTION)

cout << specVolSup + " the Encrypted File System (EFS).\n";

if (fileSystemFlags & FILE\_SUPPORTS\_EXTENDED\_ATTRIBUTES)

cout << specVolSup + " extended attributes.\n";

if (fileSystemFlags & FILE\_SUPPORTS\_HARD\_LINKS)

cout << specVolSup + " hard links. \n";

if (fileSystemFlags & FILE\_SUPPORTS\_OBJECT\_IDS)

cout << specVolSup + " object identifiers.\n";

if (fileSystemFlags & FILE\_SUPPORTS\_OPEN\_BY\_FILE\_ID)

cout << specVolSup + " open by FileID.\n";

if (fileSystemFlags & FILE\_SUPPORTS\_REPARSE\_POINTS)

cout << specVolSup + " reparse points.\n";

if (fileSystemFlags & FILE\_SUPPORTS\_SPARSE\_FILES)

cout << specVolSup + " sparse files.\n";

if (fileSystemFlags & FILE\_SUPPORTS\_TRANSACTIONS)

cout << specVolSup + " transactions.\n";

if (fileSystemFlags & FILE\_SUPPORTS\_USN\_JOURNAL)

cout << specVolSup + " update sequence number (USN) journals.\n";

if (fileSystemFlags & FILE\_UNICODE\_ON\_DISK)

cout << specVolSup + " Unicode in file names as they appear on disk.\n";

if (fileSystemFlags & FILE\_VOLUME\_IS\_COMPRESSED)

cout << specVol + " is a compressed volume, for example, a DoubleSpace volume.\n";

if (fileSystemFlags & FILE\_VOLUME\_QUOTAS)

cout << specVolSup + " disk quotas.\n";

}

long long FreeBytesAvailableToCaller;

long long TotalNumberOfBytes;

GetDiskFreeSpaceExA(drive.c\_str(), (PULARGE\_INTEGER)&FreeBytesAvailableToCaller, (PULARGE\_INTEGER)&TotalNumberOfBytes, nullptr);

cout << "\nTotal drive space: " << TotalNumberOfBytes << " bytes" << endl;

cout << "Available drive space: " << FreeBytesAvailableToCaller << " bytes" << endl;

}

// ---------- 3 -- CREATE DIRECTORY ----------

bool dirExists(const std::string& dirName\_in)

{

DWORD ftyp = GetFileAttributesA(dirName\_in.c\_str());

if (ftyp == INVALID\_FILE\_ATTRIBUTES)

{

return false; // something is wrong with your path!

}

else if (ftyp & FILE\_ATTRIBUTE\_DIRECTORY)

{

return true; // this is a directory!

}

else

{

return false; // this is not a directory!

}

}

// ---------- 3 -- GET PATH KERNEL ----------

// 'c' -- current (working) directory path

// 'a' -- absolute directory path

// 'r' -- relative directory path

// 'f' -- absolute (full) file path

// 's' -- relative (short) file path

string GetPathKernel (char localFlag)

{

char localCommit = 'n';

string localPath;

string localOldPath = currentPath;

while (localCommit != 'y')

{

if (localFlag == 'a') // 'a' means "absolute path"

{

cout << "Please, input absolute directory path:\n";

cin >> localPath;

}

else if (localFlag == 'r') // 'r' means "relative path"

{

cout << "Your current path is \"" << currentPath << "\". Please, input relative directory path:\n";

cin >> localPath;

localPath = currentPath + localPath;

}

else if (localFlag == 'f') // 'f' means "full file path"

{

cout << "Please, input absolute file path:\n";

cin >> localPath;

}

else if (localFlag == 's') // 's' means "short file path"

{

cout << "Your current path is \"" << currentPath << "\". Please, input relative file path:\n";

cin >> localPath;

localPath = currentPath + localPath;

}

else if (localFlag == 'c') // 'c' means "current"

{

cout << "Please, input current working directory path:\n";

cin >> localPath;

if (localPath.length() > 0)

{

if (localPath.c\_str()[localPath.length() - 1] != '\\')

{

localPath = localPath + "\\";

}

}

}

else // current working path by default

{

cout << "Please, input current working directory path:\n";

cin >> localPath;

}

if (localFlag != 'a' && localFlag != 'r' && localFlag != 'f' && localFlag != 's' && dirExists(localPath))

{

if (localFlag == 'c') // 'c' means "current"

{

cout << "This is your new current working path: " << localPath << "\n";

}

else // current working path by default

{

cout << "This is your new current working path: " << localPath << "\n";

}

cout << "Commit changes? [y/n]\n";

cin >> localCommit;

}

else if (localFlag == 'a' || localFlag == 'r') // for directories

{

if (localFlag == 'a') // 'a' means "absolute path"

{

cout << "This is your new absolute directory path: " << localPath << "\n";

}

else if (localFlag == 'r') // 'r' means "relative path"

{

cout << "This is your new relative directory path: " << localPath << "\n";

}

cout << "Commit changes? [y/n]\n";

cin >> localCommit;

}

else if (localFlag == 'f' || localFlag == 's') // for files

{

if (localFlag == 'f') // 'f' means "full file path"

{

cout << "This is your new absolute file path: " << localPath << "\n";

}

else if (localFlag == 's') // 's' means "shorth file path"

{

cout << "This is your new relative file path: " << localPath << "\n";

}

cout << "Commit changes? [y/n]\n";

cin >> localCommit;

}

else

{

cout << "Your new path \"" << localPath << "\"isn't valid (exist)! Try again (if no, old path will be returned)? [y/n]: " << "\n";

cin >> localCommit;

if (localCommit == 'y')

{

localCommit = 'n';

}

else if (localCommit == 'n')

{

localCommit = 'y';

localPath = localOldPath;

}

}

}

if (localFlag != 'a' && localFlag != 'r' && localFlag != 'f' && localFlag != 's')

{

currentPath = localPath;

}

// because for setting current directory it doesn't matter, but for creating/removing it matters,

// so to unificate the program, i make local path a current only if it's not for creating or deleting,

// but i return the local path (for every flags it will be right, only consist different things)

return localPath;

}

// ---------- 3 -- GET PATH SHELL ----------

string GetPathShell (char localFlagOne, char localFlagTwo, string localMessageOne, string localMessageTwo)

{

//"Path to the FILE, which ATTRIBUTES you WANT TO GET.\n"

char localFunctionFlag = '!'; // just random symbol

string localFunctionPath;

while (localFunctionFlag != localFlagOne && localFunctionFlag != localFlagTwo)

{

cout << localMessageOne << localMessageTwo;

// checking for the flag

cin >> localFunctionFlag;

if (localFunctionFlag != localFlagOne && localFunctionFlag != localFlagTwo)

{

cout << "Try again!\n";

}

}

if (localFunctionFlag == localFlagOne) // full file path situation

{

localFunctionPath = GetPathKernel(localFunctionFlag); // set new absolute path

}

else if (localFunctionFlag == localFlagTwo) // short file path situation

{

char localChange = 'n';

cout << "Do you want to change current working path? [y/n]\n";

cin >> localChange;

if (localChange == 'y')

{

currentPath = GetPathKernel('c'); // changing current directory

}

localFunctionPath = GetPathKernel(localFunctionFlag); // set new relative path

}

return localFunctionPath;

}

// ---------- 3 -- LOCAL CREATE DIRECTORY ----------

void LocalCreateRemoveDirectory (char actionCreateRemove) // if 'c' -- creating directory, if 'r' -- removing directory, creating by default

{

char localPathFlag = 'y'; // just another letter, not 'a' or 'r', so you need input it anyway

string localDirectory; // directory path you input

localDirectory = GetPathShell('a', 'r', "", "Do you want to input absolute path of the directory or relative? [a/r]\n");

if (actionCreateRemove == 'r') // 'r'

{

if (RemoveDirectory(localDirectory.c\_str()))

{

cout << "The directory \"" << localDirectory << "\" has been successfully removed!\n";

}

else

{

cout << "Something wrong! The directory \"" << localDirectory << "\" hasn't been removed!\n";

cout << "Last error code is \"" << GetLastError() << "\"\n";

}

}

else // 'c' and default

{

if (CreateDirectory(localDirectory.c\_str(), NULL))

{

cout << "The directory \"" << localDirectory << "\" has been successfully created!\n";

}

else

{

cout << "Something wrong! The directory \"" << localDirectory << "\" hasn't been created!\n";

cout << "Last error code is \"" << GetLastError() << "\"\n";

}

}

}

// ---------- 4 -- LOCAL CREATE FILE ----------

void LocalCreateFile () // A WISE FACT: THERE IS NO "OPEN FILE" FINCTION, THERE IS "CREATE FILE" FUNCTION WITH SPECIAL FLAG TO OPEN FILE!

{

int localChoose = 0; // to start a loop

unsigned long localCreationDisposition; // DWORD = unsigned long, localChoose = number that has been written in specification

char localPathFlag = 'y'; // just another letter, not 'f' or 's', so you need input it anyway

string localFilePath; // file path you input

localFilePath = GetPathShell('f', 's', "Path to the FILE, which you WANT TO CREATE (OPEN).\n", "Do you want to input absolute (full) path of the file or relative (short)? [f/s]\n");

while (localChoose < 1 || localChoose > 5)

{

// because "CREATE\_NEW" by default (1 is number for "CREATE\_NEW")

cout << "Please, choose the creation disposition (CREATE\_NEW by default):\n" << "1 -- CREATE\_NEW\n" << "2 -- CREATE\_ALWAYS\n" << "3 -- OPEN\_EXISTING\n"

<< "4 -- OPEN\_ALWAYS\n" << "5 -- TRUNCATE\_EXISTING\n"; // DWORD = unsigned long, localChoose = number that has been written in specification

cin >> localChoose;

if (localChoose < 1 || localChoose > 5)

{

cout << "Try again!\n";

}

}

switch(localChoose)

{

case 1:

localCreationDisposition = (unsigned long)localChoose; // DWORD = unsigned long, localChoose = number that has been written in specification

break;

case 2:

localCreationDisposition = (unsigned long)localChoose;

break;

case 3:

localCreationDisposition = (unsigned long)localChoose;

break;

case 4:

localCreationDisposition = (unsigned long)localChoose;

break;

case 5:

localCreationDisposition = (unsigned long)localChoose;

break;

default:

localCreationDisposition = (unsigned long)localChoose;

break;

}

// Open a handle to the file

HANDLE localFile = CreateFile(

localFilePath.c\_str(), // Filename (<path to the file>)

GENERIC\_WRITE, // Desired access (0/GENERIC\_READ/GENERIC\_WRITE)

FILE\_SHARE\_READ, // Share mode (FILE\_SHARE\_DELETE/FILE\_SHARE\_READ/FILE\_SHARE\_WRITE)

NULL, // Security attributes (NULL/<structure SECURITY\_ATTRIBUTES adress>)

(DWORD)localCreationDisposition, // Creates a new file, only if it doesn't already exist (CREATE\_ALWAYS/CREATE\_NEW/OPEN\_ALWAYS/OPEN\_EXISTING/TRUNCATE\_EXISTING)

FILE\_ATTRIBUTE\_NORMAL, // Flags and attributes (FILE\_ATTRIBUTE\_NORMAL/FILE\_ATTRIBUTE\_ARCHIVE/FILE\_ATTRIBUTE\_ENCRYPTED/FILE\_ATTRIBUTE\_SYSTEM/

// FILE\_ATTRIBUTE\_HIDDEN/FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED/FILE\_ATTRIBUTE\_OFFLINE/FILE\_ATTRIBUTE\_READONLY/FILE\_ATTRIBUTE\_SYSTEM/FILE\_ATTRIBUTE\_TEMPORARY/

// FILE\_FLAG\_BACKUP\_SEMANTICS/FILE\_FLAG\_DELETE\_ON\_CLOSE/FILE\_FLAG\_NO\_BUFFERING/FILE\_FLAG\_OPEN\_NO\_RECALL/FILE\_FLAG\_OPEN\_REPARSE\_POINT/FILE\_FLAG\_OVERLAPPED/

// FILE\_FLAG\_POSIX\_SEMANTICS/FILE\_FLAG\_RANDOM\_ACCESS/FILE\_FLAG\_SEQUENTIAL\_SCAN/FILE\_FLAG\_WRITE\_THROUGH/SECURITY\_ANONYMOUS/SECURITY\_CONTEXT\_TRACKING/

// SECURITY\_DELEGATION/SECURITY\_EFFECTIVE\_ONLY/SECURITY\_IDENTIFICATION/SECURITY\_IMPERSONATION)

NULL); // Template file handle (NULL/<template file descripter>)

if (localFile != INVALID\_HANDLE\_VALUE)

{

cout << "The file \"" << localFilePath << "\" has been successfully created (opened, rewrited)!\n";

CloseHandle(localFile);

}

else

{

cout << "Something wrong! The file \"" << localFilePath << "\" hasn't been created (opened, rewrited)!\n";

cout << "Last error code is \"" << GetLastError() << "\"\n";

}

}

// ---------- 5 -- LOCAL COPY MOVE FILE ----------

void LocalCopyMoveFile (char actionCopyMove) // 'c' for copy, 'm' for moving, 'e' for extended moving (MoveFileEx) ATTENTION: 'c' -- copy -- is default if there is other letter!

{

// specification of "CopyFile"

/\*BOOL CopyFile(

LPCTSTR lpExistingFileName, // current file you want to copy

LPCTSTR lpNewFileName, // new file, where you wnat to copy the old one

BOOL bFailIfExists // TRUE means STOP IF NEW FILE EXIST, FALSE means OWERWRITE FILE ANYWAY

);\*/

int localChoose = 0; // to start a loop

bool localFailIfExists; // for existing file reaction

char localPathFlag = 'y'; // just another letter, not 'f' or 's', so you need input it anyway

char localSameNameFlag = 'a'; // random letter

string localOldFilePath; // old (copied) file path you input

string localNewFilePath; // new (pasted) file path you input

// OLD FILE PATH INPUT (INCLUDING SITUATIONS "COPY" AND "MOVE")

if (actionCopyMove == 'm' || actionCopyMove == 'e') // 'm' is for "moving", 'e' is for "extended moving"

{

localOldFilePath = GetPathShell('f', 's', "Path to the FILE, which you WANT TO MOVE (CUT).\n", "Do you want to input absolute (full) path of the file or relative (short)? [f/s]\n");

}

else // 'c' is for "copy", "copy" is default

{

localOldFilePath = GetPathShell('f', 's', "Path to the FILE, which you WANT TO COPY.\n", "Do you want to input absolute (full) path of the file or relative (short)? [f/s]\n");

}

// SAME NAME SITUATION

while (localSameNameFlag != 'y' && localSameNameFlag != 'n')

{

cout << "Do you want the SAME NAME AS COPIED/CUT file (including extension)? [y/n]\n";

// checking for the flag

cin >> localSameNameFlag;

if (localSameNameFlag != 'y' && localSameNameFlag != 'n') // 'n' -- no -- by default

{

cout << "Try again!\n";

}

}

// NEW FILE PATH INPUT

//localPathFlag = 'y'; // random letter again to prevent ignoring if construction

if (localSameNameFlag == 'y') // same name of the file -- choosing full directory path instead of the filename

{

string localFilename = string(localOldFilePath);

// Remove directory if present.

// Do this before extension removal incase directory has a period character.

const size\_t last\_slash\_idx = localFilename.find\_last\_of("\\/");

if (std::string::npos != last\_slash\_idx)

{

localFilename.erase(0, last\_slash\_idx + 1);

}

// Remove extension if present.

/\*const size\_t period\_idx = localFilename.rfind('.');

if (std::string::npos != period\_idx)

{

localFilename.erase(period\_idx);

}\*/

if (actionCopyMove == 'm' || actionCopyMove == 'e') // 'm' is for "moving", 'e' is for "extended moving"

{

localNewFilePath = GetPathShell('a', 'r', "Path to the NEW DIRECTORY, where you WANT to MOVE (PASTE) the CUT FILE.\n", "Do you want to input absolute path of the directory or relative? [a/r]\n") + localFilename;

}

else // 'c' is for "copy", "copy" is DEFAULT

{

localNewFilePath = GetPathShell('a', 'r', "Path to the NEW DIRECTORY, where you WANT to PASTE the COPIED FILE.\n", "Do you want to input absolute path of the directory or relative? [a/r]\n") + localFilename;

}

}

else // 'n', different (as user choosed) name of the file -- choosing full file path (not the same name is DEFAULT)

{

if (actionCopyMove == 'm' || actionCopyMove == 'e') // 'm' is for "moving", 'e' is for "extended moving"

{

localNewFilePath = GetPathShell('f', 's', "Path to the NEW FILE, where you WANT to MOVE (PASTE) the CUT FILE.\n", "Do you want to input absolute (full) path of the file or relative (short)? [f/s]\n");

}

else // 'c' is for "copy", "copy" is DEFAULT

{

localNewFilePath = GetPathShell('f', 's', "Path to the NEW FILE, where you WANT to PASTE the COPIED FILE.\n", "Do you want to input absolute (full) path of the file or relative (short)? [f/s]\n");

}

}

// COPY and MOVE

if (actionCopyMove == 'c' || (actionCopyMove != 'm' && actionCopyMove != 'e')) // copy

{

// IF NEW FILE EXISTS

while (localChoose < 1 || localChoose > 2)

{

// because "CREATE\_NEW" by default (1 is number for "CREATE\_NEW")

cout << "Please, choose the possible outcome for you IF NEW FILE EXISTS:\n" << "1 -- DON'T PASTE AND DON'T REWRITE ANY FILE (TRUE)\n"

<< "2 -- REWRITE THE FILE (FALSE)\n";

cin >> localChoose;

if (localChoose < 1 || localChoose > 2)

{

cout << "Try again!\n";

}

}

switch(localChoose)

{

case 1:

localFailIfExists = true;

break;

case 2:

localFailIfExists = false;

break;

default:

localFailIfExists = true; // true -- file rewriting protection -- by default

break;

}

// don't forget about c\_str()!

if (CopyFile(localOldFilePath.c\_str(), localNewFilePath.c\_str(), localFailIfExists)) // copy file and watching the result immediatly

{

cout << "The file \"" << localOldFilePath << "\" has been successfully copied to file \"" << localNewFilePath << "\"!\n";

}

else

{

cout << "Something wrong! The file \"" << localOldFilePath << "\" hasn't been copied to file \"" << localNewFilePath << "\"!\n";

cout << "Last error code is \"" << GetLastError() << "\"\n";

}

}

else if (actionCopyMove == 'm') // move

{

// don't forget about c\_str()!

if (MoveFile(localOldFilePath.c\_str(), localNewFilePath.c\_str())) // move file and watching the result immediatly

{

cout << "The file \"" << localOldFilePath << "\" has been successfully moved to file \"" << localNewFilePath << "\"!\n";

}

else

{

cout << "Something wrong! The file \"" << localOldFilePath << "\" hasn't been moved to file \"" << localNewFilePath << "\"!\n";

cout << "Last error code is \"" << GetLastError() << "\"\n";

}

}

else if (actionCopyMove == 'e') // extended move

{

string localChooseTwo = "2"; // because default

// next parameters presented in decimal code, but can be translated at once (for example "111111" is all of the flags (but all is uncorrect)!)

// MOVEFILE\_COPY\_ALLOWED = 2 (copy, than delete old, normal work), cannot be used with MOVEFILE\_DELAY\_UNTIL\_REBOOT

// MOVEFILE\_CREATE\_HARDLINK = 16 ("Reserved for future use", i don't know what does it means)

// MOVEFILE\_DELAY\_UNTIL\_REBOOT = 4 (waiting util reboot), cannot be used with MOVEFILE\_COPY\_ALLOWED

// MOVEFILE\_FAIL\_IF\_NOT\_TRACKABLE = 32 ("function fails, if the file is the lik source")

// MOVEFILE\_REPLACE\_EXISTING = 1 (replacing file)

// MOVEFILE\_WRITE\_THROUGH = 8 (doesn't return anything until it's ACTUALLY MOVE SOMETHING!)

unsigned long inFunctionNumber = 0;

while (inFunctionNumber == 0)

{

// because "CREATE\_NEW" by default (1 is number for "CREATE\_NEW")

cout << "Please, choose the possible flags for moving file (you CAN CHOOSE MANY -- JUST SPLIT NUMBERS BY SPACE):\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";

fflush(stdin);

std::getline(std::cin, localChooseTwo);

// spit the string

std::string s = string(localChooseTwo);

std::string delimiter = " ";

int i = 0;

size\_t pos = 0;

std::string token;

std::vector<string> v;

std::vector<int> vect{1, 2, 4, 8, 16, 32};

while ((pos = s.find(delimiter)) != std::string::npos)

{

int tmpNumber = 0;

token = s.substr(0, pos);

v.push\_back(token);

tmpNumber = std::stoi(token);

if (std::find(vect.begin(), vect.end(), tmpNumber) != vect.end())

{

inFunctionNumber = inFunctionNumber + tmpNumber;

vect.erase(std::remove(vect.begin(), vect.end(), tmpNumber), vect.end());

}

//std::cout << token << std::endl;

s.erase(0, pos + delimiter.length());

}

int newTMPNumber = std::stoi(s);

if (std::find(vect.begin(), vect.end(), newTMPNumber) != vect.end())

{

inFunctionNumber = inFunctionNumber + newTMPNumber;

vect.erase(std::remove(vect.begin(), vect.end(), newTMPNumber), vect.end());

}

//cout << inFunctionNumber;

//std::cout << s << std::endl;

// end split of the string

if (inFunctionNumber == 0)

{

cout << "Try again!\n";

}

}

// don't forget about c\_str()!

if (MoveFileEx(localOldFilePath.c\_str(), localNewFilePath.c\_str(), (DWORD)inFunctionNumber)) // extended move file and watching the result immediatly

{

cout << "The file \"" << localOldFilePath << "\" has been successfully moved to file \"" << localNewFilePath << "\"!\n";

}

else

{

cout << "Something wrong! The file \"" << localOldFilePath << "\" hasn't been moved to file \"" << localNewFilePath << "\"!\n";

cout << "Last error code is \"" << GetLastError() << "\"\n"; // here i need to insert last error text string

}

}

}

// ---------- 6 -- LOCAL GET FILE ATTRIBUTES ----------

void LocalGetFileAttributes ()

{

// specification of "GetFileAttributesA"

/\*DWORD GetFileAttributesA(

LPCSTR lpFileName // file name, which i want to get the file attributes

);\*/

DWORD localFileAttributes = 0;

string localFilePath; // file path you input

// FILE PATH INPUT

localFilePath = GetPathShell('f', 's', "Path to the FILE or DIRECTORY, which ATTRIBUTES you WANT TO GET.\n", "Do you want to input absolute (full) path of the file (directory) or relative (short)? [f/s]\n");

localFileAttributes = GetFileAttributes(localFilePath.c\_str());

cout << "File \"" << localFilePath << "\" attributes:\n";

if (localFileAttributes & FILE\_ATTRIBUTE\_ARCHIVE)

{

cout << "Archive (FILE\_ATTRIBUTE\_ARCHIVE)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_COMPRESSED)

{

cout << "Compressed (FILE\_ATTRIBUTE\_COMPRESSED)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_DEVICE)

{

cout << "Device (FILE\_ATTRIBUTE\_DEVICE)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY)

{

cout << "Directory (FILE\_ATTRIBUTE\_DIRECTORY)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_ENCRYPTED)

{

cout << "Encrypted (FILE\_ATTRIBUTE\_ENCRYPTED)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_HIDDEN)

{

cout << "Hidden (FILE\_ATTRIBUTE\_HIDDEN)\n";

}

/\*if (localFileAttributes & FILE\_ATTRIBUTE\_INTEGRITY\_STREAM)

{

cout << "Data stream configured with integrity (FILE\_ATTRIBUTE\_INTEGRITY\_STREAM)\n";

}\*/

if (localFileAttributes & FILE\_ATTRIBUTE\_NORMAL)

{

cout << "Normal (FILE\_ATTRIBUTE\_NORMAL)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED)

{

cout << "Not indexed (FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED)\n";

}

/\*if (localFileAttributes & FILE\_ATTRIBUTE\_NO\_SCRUB\_DATA)

{

cout << "Data stream not to be read by the data integrity scanner (FILE\_ATTRIBUTE\_NO\_SCRUB\_DATA)\n";

}\*/

if (localFileAttributes & FILE\_ATTRIBUTE\_OFFLINE)

{

cout << "Don't avaliable immediatly (FILE\_ATTRIBUTE\_OFFLINE)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_READONLY)

{

cout << "Read-only (FILE\_ATTRIBUTE\_READONLY)\n";

}

/\*if (localFileAttributes & FILE\_ATTRIBUTE\_RECALL\_ON\_DATA\_ACCESS)

{

cout << "Data is not fully presented locally (FILE\_ATTRIBUTE\_RECALL\_ON\_DATA\_ACCESS)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_RECALL\_ON\_OPEN)

{

cout << "Data hasn't physical representation on system (FILE\_ATTRIBUTE\_RECALL\_ON\_OPEN)\n";

}\*/

if (localFileAttributes & FILE\_ATTRIBUTE\_REPARSE\_POINT)

{

cout << "Reparse point/representation link (FILE\_ATTRIBUTE\_REPARSE\_POINT)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_SPARSE\_FILE)

{

cout << "Sparse file (FILE\_ATTRIBUTE\_SPARSE\_FILE)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_SYSTEM)

{

cout << "System used (FILE\_ATTRIBUTE\_SYSTEM)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_TEMPORARY)

{

cout << "Temporary storage (FILE\_ATTRIBUTE\_TEMPORARY)\n";

}

if (localFileAttributes & FILE\_ATTRIBUTE\_VIRTUAL)

{

cout << "Reserved for system (FILE\_ATTRIBUTE\_VIRTUAL)\n";

}

}

// ---------- 6 -- LOCAL SET FILE ATTRIBUTES ----------

void LocalSetFileAttributes ()

{

// specification of "SetFileAttributesA"

/\*BOOL SetFileAttributesA(

LPCSTR lpFileName, // filename

DWORD dwFileAttributes //attributes

);\*/

DWORD localFileAttributes = 0;

string localFilePath; // file path you input

// FILE PATH INPUT

localFilePath = GetPathShell('f', 's', "Path to the FILE or DIRECTORY, which ATTRIBUTES you WANT TO SET.\n", "Do you want to input absolute (full) path of the file (directory) or relative (short)? [f/s]\n");

localFileAttributes = GetFileAttributes(localFilePath.c\_str());

string localChooseTwo = "128";

unsigned long inFunctionNumber = 0;

while (inFunctionNumber == 0)

{

// because "CREATE\_NEW" by default (1 is number for "CREATE\_NEW")

cout << "Please, choose the possible attributes for the file/directory (you CAN CHOOSE MANY -- JUST SPLIT NUMBERS BY SPACE):\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";

fflush(stdin);

std::getline(std::cin, localChooseTwo);

// spit the string

std::string s = string(localChooseTwo);

std::string delimiter = " ";

int i = 0;

size\_t pos = 0;

std::string token;

std::vector<string> v;

std::vector<int> vect{1, 2, 4, 32, 128, 256, 4096, 8192};

while ((pos = s.find(delimiter)) != std::string::npos)

{

int tmpNumber = 0;

token = s.substr(0, pos);

v.push\_back(token);

tmpNumber = std::stoi(token);

if (std::find(vect.begin(), vect.end(), tmpNumber) != vect.end())

{

inFunctionNumber = inFunctionNumber + tmpNumber;

vect.erase(std::remove(vect.begin(), vect.end(), tmpNumber), vect.end());

}

//std::cout << token << std::endl;

s.erase(0, pos + delimiter.length());

}

int newTMPNumber = std::stoi(s);

if (std::find(vect.begin(), vect.end(), newTMPNumber) != vect.end())

{

inFunctionNumber = inFunctionNumber + newTMPNumber;

vect.erase(std::remove(vect.begin(), vect.end(), newTMPNumber), vect.end());

}

//cout << inFunctionNumber;

//std::cout << s << std::endl;

// end split of the string

if (inFunctionNumber == 0)

{

cout << "Try again!\n";

}

}

if (SetFileAttributes(localFilePath.c\_str(), (DWORD) inFunctionNumber))

{

cout << "The file's (directory's) \"" << localFilePath << "\" attributes has been successfully changed to:\n";

cout << "File \"" << localFilePath << "\" attributes:\n";

if (inFunctionNumber & FILE\_ATTRIBUTE\_ARCHIVE)

{

cout << "Archive (FILE\_ATTRIBUTE\_ARCHIVE)\n";

}

if (inFunctionNumber & FILE\_ATTRIBUTE\_COMPRESSED)

{

cout << "Compressed (FILE\_ATTRIBUTE\_COMPRESSED)\n";

}

if (inFunctionNumber & FILE\_ATTRIBUTE\_DEVICE)

{

cout << "Device (FILE\_ATTRIBUTE\_DEVICE)\n";

}

if (inFunctionNumber & FILE\_ATTRIBUTE\_DIRECTORY)

{

cout << "Directory (FILE\_ATTRIBUTE\_DIRECTORY)\n";

}

if (inFunctionNumber & FILE\_ATTRIBUTE\_ENCRYPTED)

{

cout << "Encrypted (FILE\_ATTRIBUTE\_ENCRYPTED)\n";

}

if (inFunctionNumber & 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 (inFunctionNumber & FILE\_ATTRIBUTE\_NORMAL)

{

cout << "Normal (FILE\_ATTRIBUTE\_NORMAL)\n";

}

if (inFunctionNumber & 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 (inFunctionNumber & FILE\_ATTRIBUTE\_OFFLINE)

{

cout << "Don't avaliable immediatly (FILE\_ATTRIBUTE\_OFFLINE)\n";

}

if (inFunctionNumber & 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 (inFunctionNumber & FILE\_ATTRIBUTE\_REPARSE\_POINT)

{

cout << "Reparse point/representation link (FILE\_ATTRIBUTE\_REPARSE\_POINT)\n";

}

if (inFunctionNumber & FILE\_ATTRIBUTE\_SPARSE\_FILE)

{

cout << "Sparse file (FILE\_ATTRIBUTE\_SPARSE\_FILE)\n";

}

if (inFunctionNumber & FILE\_ATTRIBUTE\_SYSTEM)

{

cout << "System used (FILE\_ATTRIBUTE\_SYSTEM)\n";

}

if (inFunctionNumber & FILE\_ATTRIBUTE\_TEMPORARY)

{

cout << "Temporary storage (FILE\_ATTRIBUTE\_TEMPORARY)\n";

}

if (inFunctionNumber & FILE\_ATTRIBUTE\_VIRTUAL)

{

cout << "Reserved for system (FILE\_ATTRIBUTE\_VIRTUAL)\n";

}

}

else

{

cout << "Something wrong! The file's (directory's) \"" << localFilePath << "\" attributes hasn't been changed\n";

cout << "Last error code is \"" << GetLastError() << "\"\n"; // here i need to insert last error text string

}

}

// ---------- 6 -- LOCAL GET FILE INFORMATION BY HANDLE ----------

void LocalGetFileInformationByHandle ()

{

// specification of "GetFileInformationByHandle"

/\*BOOL GetFileInformationByHandle(

HANDLE hFile, // path to the handle

LPBY\_HANDLE\_FILE\_INFORMATION lpFileInformation // file information

);\*/

string localFilePath = GetPathShell('f', 's', "Path to the FILE or DIRECTORY, which ATTRIBUTES you WANT TO GET.\n", "Do you want to input absolute (full) path of the file (directory) or relative (short)? [f/s]\n");

HANDLE hFile = CreateFile(localFilePath.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\_INTEGRITY\_STREAM)

{

cout << "Data stream configured with integrity (FILE\_ATTRIBUTE\_INTEGRITY\_STREAM)\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\_NO\_SCRUB\_DATA)

{

cout << "Data stream not to be read by the data integrity scanner (FILE\_ATTRIBUTE\_NO\_SCRUB\_DATA)\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\_RECALL\_ON\_DATA\_ACCESS)

{

cout << "Data is not fully presented locally (FILE\_ATTRIBUTE\_RECALL\_ON\_DATA\_ACCESS)\n";

}

if (localAttributes & FILE\_ATTRIBUTE\_RECALL\_ON\_OPEN)

{

cout << "Data hasn't physical representation on system (FILE\_ATTRIBUTE\_RECALL\_ON\_OPEN)\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";

/\*DWORD localAttributes = lpFileInformation->dwFileAttributes;

//localCreationTime;

DWORD localAccessTime = (lpFileInformation->ftLastAccessTime).dwLowDateTime;

//localChangeTime;

DWORD localVolumeSerialNumber = lpFileInformation->dwVolumeSerialNumber;

DWORD localSizeHigh = lpFileInformation->nFileSizeHigh;

DWORD localSizeLow = lpFileInformation->nFileSizeLow;

DWORD localNumberOfLinks = lpFileInformation->nNumberOfLinks;

DWORD localIndexHigh = lpFileInformation->nFileIndexHigh;

DWORD localIndexLow = lpFileInformation->nFileIndexLow;\*/

/\*cout << localAttributes;

cout << localAccessTime;

cout << localVolumeSerialNumber;

cout << localSizeHigh;

cout << localSizeLow;

cout << localNumberOfLinks;

cout << localIndexHigh;

cout << localIndexLow;\*/

CloseHandle(hFile);

}

// ---------- 6 -- LOCAL GET FILE INFORMATION BY HANDLE ----------

void GetFileTime ()

{

string localFilePath = GetPathShell('f', 's', "Path to the FILE get time.\n", "Do you want to input absolute (full) path of the file (directory) or relative (short)? [f/s]\n");

HANDLE hFile = CreateFile(localFilePath.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))

{

//FILETIME ft;

//ft.dwHighDateTime = creationTime.dwHighDateTime;

//ft.dwLowDateTime = creationTime.dwLowDateTime;

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);

}

// ---------- 6 -- LOCAL GET FILE INFORMATION BY HANDLE ----------

void SetFileTime ()

{

string localFilePath = GetPathShell('f', 's', "Path to the FILE set time.\n", "Do you want to input absolute (full) path of the file (directory) or relative (short)? [f/s]\n");

HANDLE hFile = CreateFile(localFilePath.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: 2021-10-03 18:29:40.152\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);

}

## 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.

# 3. Копирование файла с помощью перекрывающих операций ввода-вывода

## 3.1. Создание и запуск консольного приложения

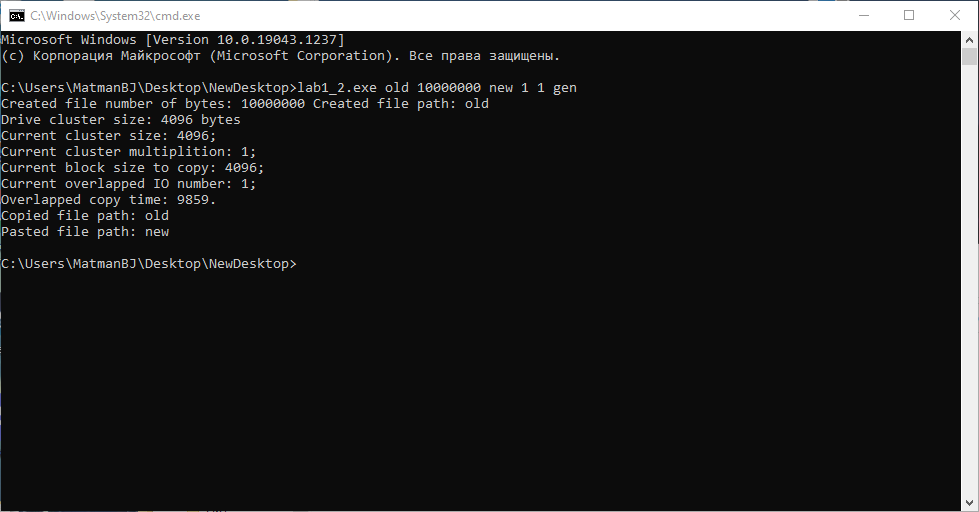


Рисунок 26: Запуск теста времени копирования

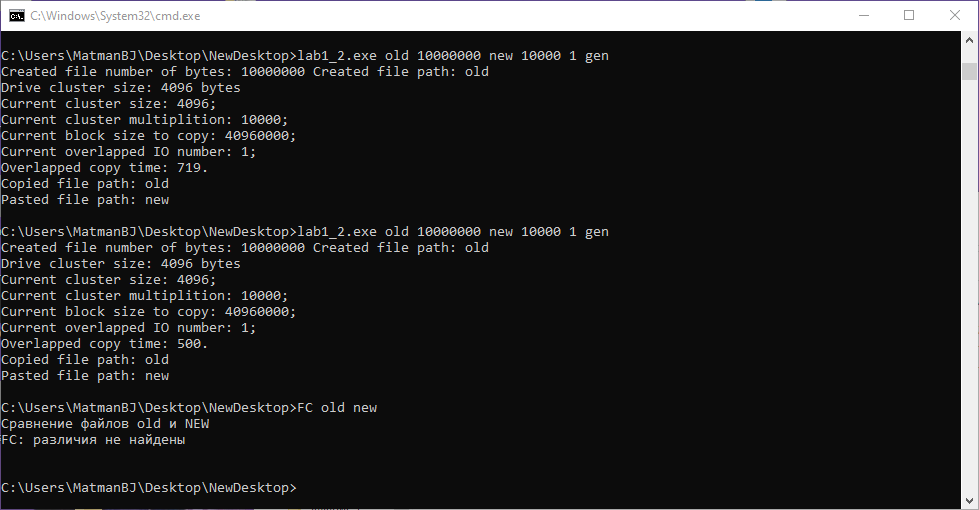


Рисунок 27: Использование команды FC для проверки результата копирования

## 3.2. Проверка приложения на разных размерах копируемых блоков

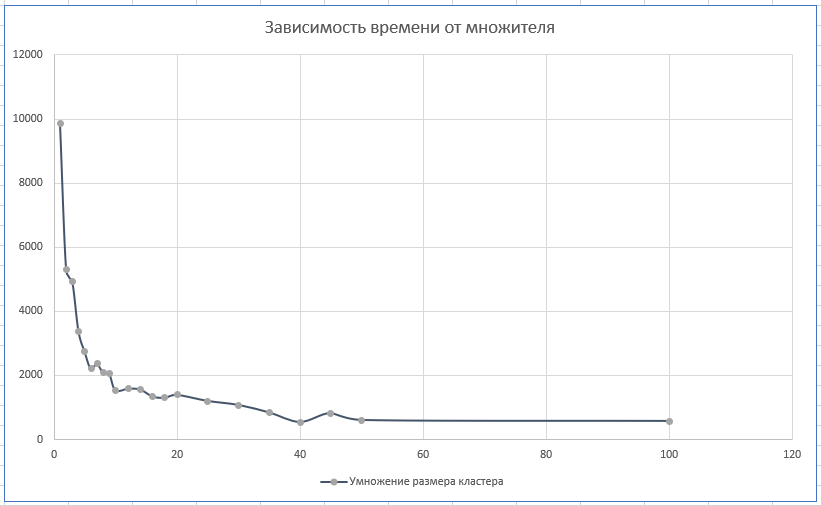


Рисунок 28: Зависимость времени от множителя размера кластера

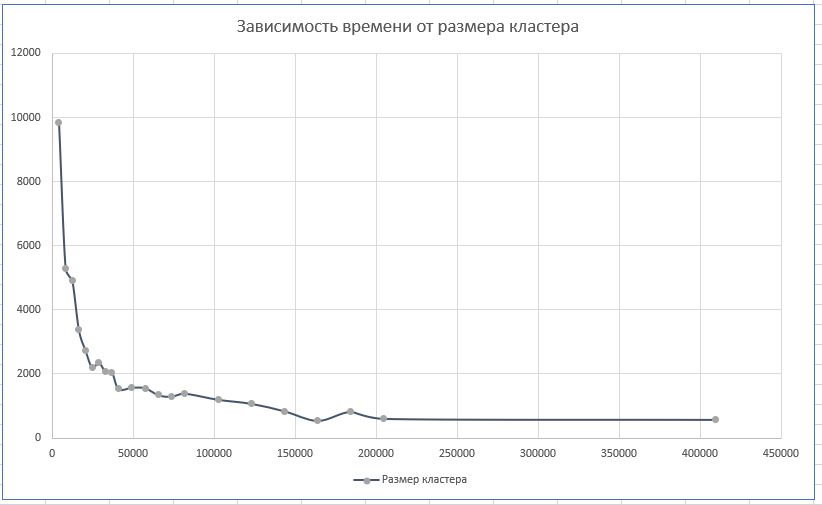


Рисунок 29: Зависимость времени от размера кластера

В работе было проведено 28 замеров с разным шагом, который постепенно увеличивался. Было учтено 22 замера до множителя, равного 100, поскольку после него значения были приближены к пределу. Далее приведены 2 графика, иллюстрирующие изменение времени копирования с 1 перекрывающей операцией ввода-вывода для разного размера копируемого блока, кратного размеру кластера на диске.

Как можно заметить, с увеличением размера копируемого блока время копирования уменьшалось. При размере блока, в 10 раз превышающего изначальный размер, спад времени копирования замедлился, а уже при значении 100 практически не менялся.

## 3.3. Проверка приложения на разном числе операция ввода-вывода

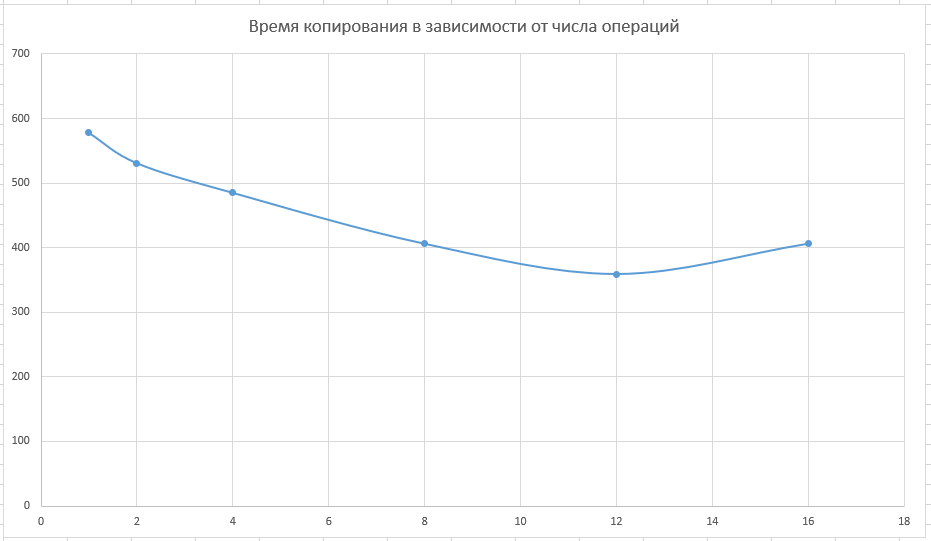


Рисунок 30: Зависимость времени от числа операций

Из результатов эксперимента можно понять, что наиболее эффективным количеством операций ввода-вывода было 12, однако на графике видно уменьшение времени копирования при увеличении количества операций, что говорит об возможной эффективности дальнейшего их увеличения.

## 3.4. Исходный код программы

#include <iostream> // for everything

#include <windows.h> // for API

#include <string> // for tring usage

#include <cstdlib>

#include <ctime>

#include <fstream>

#include <cmath>

using namespace std;

LARGE\_INTEGER shiftRead; // read structure

LARGE\_INTEGER shiftWrite; // write structure

int LocalFileGenerator (string localOldFilePath, unsigned long long localBytesRequest);

void CopyPaste (string localOldFilePath, string localNewFilePath);

DWORD PreparingCopyPaste(string localOldFilePath, string localNewFilePath, unsigned long long localBlockSize, unsigned long long localOverlappedIOSize);

void LocalReadWrite(long long fileSize, DWORD blockSize, int localOperationsCounter, OVERLAPPED\* overlappeds, CHAR\*\* buffer, HANDLE fileHandle, char f);

DWORD LocalDriveSectorSize ();

void CALLBACK CompletionRoutine(DWORD dwErrorCode, DWORD dwNumberOfBytesTransfered, LPOVERLAPPED lpOverlapped);

// HOW TO START

// <name.exe> <inpit file name> <bytes> <output file name> <number of cluster (sector\*bytespersec) multiplition> <number of operations overlapped> <"gen" for generting file, other for file>

unsigned long long callback;

DWORD copyTime;

unsigned long long bs = 1; // umber of block

unsigned long long oios = 1; // number of operations for Overlapped I/O

// ---------- MAIN ----------

int main(int argc, char \*\*argv)

{

if (argc == 7)

{

const unsigned long oldFileBytes = atoi(argv[2]); // size

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

const string newFilePath(argv[3]); // new file

bs = atoi(argv[4]); // multiplitions

oios = atoi(argv[5]); // operations

string gen = "gen";

if (strcmp(argv[6], gen.c\_str()) == 0)

{

LocalFileGenerator (oldFilePath, oldFileBytes);

}

cout << "Created file number of bytes: " << oldFileBytes << " Created file path: " << oldFilePath << "\n";

CopyPaste(oldFilePath, newFilePath);

cout << "Copied file path: " << oldFilePath << "\nPasted file path: " << newFilePath << "\n";

return 0;

}

else

{

cout << "Incorrect start of executable file. Please, check your flags!\n";

return -1;

}

}

// ---------- FILE GENERATION FOR COPY TESTS ----------

int LocalFileGenerator (string localOldFilePath, unsigned long long localBytesRequest)

{

unsigned long long i;

ofstream localFile (localOldFilePath, ios :: out | ios :: binary | ios :: app);

srand (time(NULL));

char localByte[10];

for (i = 0; i < localBytesRequest; i++)

{

localByte[0] = (unsigned char)(rand() % 256);

localFile.write (localByte, sizeof(localByte));

}

localFile.close();

return 0;

}

// ---------- COPY AND PASTE MAIN PROCESS ----------

void CopyPaste (string localOldFilePath, string localNewFilePath)

{

// ATTENTION: I'VE CHANGED SECTOR SIZE TO CLUSTER SIZE, BUT I DIN'T CHANGE VARS AND FUCNTIONS NAMES

DWORD localSectorSize = LocalDriveSectorSize ();

unsigned long long localBlockSize; // size of the data block I will copy

cout << "Drive cluster size: " << localSectorSize << " bytes\n";

localBlockSize = localSectorSize\*bs;

copyTime = 0;

copyTime = PreparingCopyPaste(localOldFilePath, localNewFilePath, localBlockSize, oios);

cout << "Current cluster size: " << localSectorSize << ";\n"

<< "Current cluster multiplition: " << bs << ";\n"

<< "Current block size to copy: " << localBlockSize << ";\n"

<< "Current overlapped IO number: " << oios << ";\n"

<< "Overlapped copy time: " << copyTime << ".\n";

}

// ---------- PREPARING FOR COPY AND PASTE ACTIONS ----------

DWORD PreparingCopyPaste(string localOldFilePath, string localNewFilePath, unsigned long long localBlockSize, unsigned long long localOverlappedIOSize)

{

HANDLE localOldFileHandle = CreateFileA(localOldFilePath.c\_str(), GENERIC\_READ, 0, NULL, OPEN\_EXISTING, FILE\_FLAG\_OVERLAPPED | FILE\_FLAG\_NO\_BUFFERING, NULL); // copied file path

HANDLE localNewFileHandle = CreateFileA(localNewFilePath.c\_str(), GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, FILE\_FLAG\_OVERLAPPED | FILE\_FLAG\_NO\_BUFFERING, NULL); // new file path

DWORD lpFileSizeHigh; // lpdword filesize high

DWORD getFileSize;

//unsigned long long fileSize;

if (localOldFileHandle == NULL || localOldFileHandle == INVALID\_HANDLE\_VALUE || localNewFileHandle == NULL || localNewFileHandle == INVALID\_HANDLE\_VALUE)

{

cout << "Problem with opening files!\n";

cout << "\nError message: " << GetLastError() << "\n";

}

else

{

DWORD blockSize = localBlockSize;

getFileSize = GetFileSize(localOldFileHandle, &lpFileSizeHigh);

DWORD high = 0;

LARGE\_INTEGER fileSizeStruct; // where file size will be wrote

long long fileSize; // number where we write

GetFileSizeEx(localOldFileHandle, &fileSizeStruct); // file size (for input as LARGE\_INTEGER we use it!!!)

fileSize = fileSizeStruct.QuadPart; // uniting fields one number

long long currentSize = fileSize;

CHAR\*\* buffer = new CHAR\*[localOverlappedIOSize]; // buffer of the data

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

{

buffer[i] = new CHAR[(int)blockSize];

}

OVERLAPPED\* over\_1 = new OVERLAPPED[localOverlappedIOSize]; // var for handling pointers (starting bytes)

OVERLAPPED\* over\_2 = new OVERLAPPED[localOverlappedIOSize]; // new file

shiftRead.QuadPart = 0; // how many rode

shiftWrite.QuadPart = 0; // how many write

copyTime = GetTickCount(); // time counting

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

{

over\_1[i].Offset = over\_2[i].Offset = shiftRead.LowPart; // FIRST PART OF THE STRUCTURE

over\_1[i].OffsetHigh = over\_2[i].OffsetHigh = shiftRead.HighPart; // SEONDN PART (^$ bit

shiftRead.QuadPart += blockSize;

shiftWrite.QuadPart += blockSize;

}

do

{

LocalReadWrite(currentSize, blockSize, localOverlappedIOSize, over\_1, buffer, localOldFileHandle, 'r');

LocalReadWrite(currentSize, blockSize, localOverlappedIOSize, over\_2, buffer, localNewFileHandle, 'w');

currentSize -= (long long)(blockSize\*localOverlappedIOSize);

}

while (currentSize > 0);

copyTime = GetTickCount() - copyTime; // time counting

SetFilePointerEx(localNewFileHandle, fileSizeStruct, NULL, FILE\_BEGIN);

SetEndOfFile(localNewFileHandle);

}

// Checking handle and closing the file

if (localOldFileHandle != NULL && localOldFileHandle != INVALID\_HANDLE\_VALUE) // old file checking

{

if (CloseHandle(localOldFileHandle) == false)

{

cout << "EROOR WHILE CLOSING FILE \"" << localOldFileHandle << "\". " << endl;

}

}

if (localNewFileHandle != NULL && localNewFileHandle != INVALID\_HANDLE\_VALUE) // new file checking

{

if (CloseHandle(localNewFileHandle) == false)

{

cout << "ERROR WHILE CLOSING FILE \"" << localNewFileHandle << "\". " << endl;

}

}

return copyTime;

}

// ---------- LOCAL READ WRITE ----------

void LocalReadWrite(long long fileSize, DWORD blockSize, int localOperationsCounter, OVERLAPPED\* overlappeds, CHAR\*\* buffer, HANDLE fileHandle, char f)

{

int operations\_counter = 0;

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

{

if (fileSize > 0)

{

operations\_counter = operations\_counter + 1;

if (f == 'r')

{

ReadFileEx(fileHandle, buffer[i], blockSize, &overlappeds[i], CompletionRoutine);

}

else if (f == 'w')

{

WriteFileEx(fileHandle, buffer[i], blockSize, &overlappeds[i], CompletionRoutine);

}

fileSize -= blockSize;

}

}

while (callback < operations\_counter)

{

SleepEx(-1, true);

}

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

{

if (f == 'r')

{

overlappeds[i].Offset = shiftRead.LowPart;

overlappeds[i].OffsetHigh = shiftRead.HighPart;

shiftRead.QuadPart += blockSize;

}

else if (f == 'w')

{

overlappeds[i].Offset = shiftWrite.LowPart;

overlappeds[i].OffsetHigh = shiftWrite.HighPart;

shiftWrite.QuadPart += blockSize;

}

}

callback = 0;

}

// ---------- GET DRIVE SECTOR SIZE ----------

DWORD LocalDriveSectorSize ()

{

DWORD localSectorsPerCluster;

DWORD localNumberOfClusters = -1;

DWORD localBytesPerSector = -1;

DWORD localNumberOfFreeClusters = -1;

if (GetDiskFreeSpaceA(NULL, &localSectorsPerCluster, &localBytesPerSector, &localNumberOfFreeClusters, &localNumberOfClusters) == false)

{

cout << "\nERROR GETTING DRIVE SECTORS\n";

}

return localBytesPerSector\*localSectorsPerCluster;

}

// ---------- AFTER READING FILE NEED TO MAKE THIS FUNCTION ----------

void CALLBACK CompletionRoutine (DWORD dwErrorCode, DWORD dwNumberOfBytesTransfered, LPOVERLAPPED lpOverlapped)

{

callback = callback + 1;

}

## 3.5. Выводы

В ходе работы было исследовано копирование файла с помощью операций перекрывающегося ввода-вывода. В эксперименте было обнаружено, что при увеличении размера копируемого блока или при увеличении числа перекрывающих операций заметно снижается скорость копирования файла, в частности, стократное увеличение размера копируемого блока и 12 операций перекрывающего ввода-вывода показывали лучший результат по времени выполнения. Стоит учитывать и то, что на ход эксперимента могли повлиять такие факторы, загрузка процессора, файловая система и скорость записи на накопитель. Таким образом и было реализовано копирование с помощью операций перекрывающегося ввода-вывода.

# 4. Список использованных источников

1. Операционные системы: электронные методические указания к лабораторным работам / Сост.: А. В. Тимофеев. СПб.: Изд-во СПбГЭТУ

«ЛЭТИ», 2016.

2. Курс «Операционные системы» в образовательной онлайн-системе Google Класс [сайт]. URL: <https://classroom.google.com/c/Mzg3ODc4NDE5MDU4>.