

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра ВТ**

**ОТЧЕТ**  
**по лабораторной работе №1**  
**по дисциплине «Операционные системы»**  
**Тема: Управление файловой системой**

Студент гр. 8305

\_\_\_\_\_

Родионов Г.В.

Преподаватель

\_\_\_\_\_

Тимофеев А.В.

Санкт-Петербург

2020

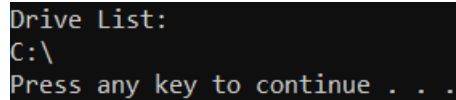
## 1 Цель работы

Целью работы является исследование управления файловой системой с помощью WinAPI.

## 2 Задание 1

Управление дисками, каталогами и файлами.

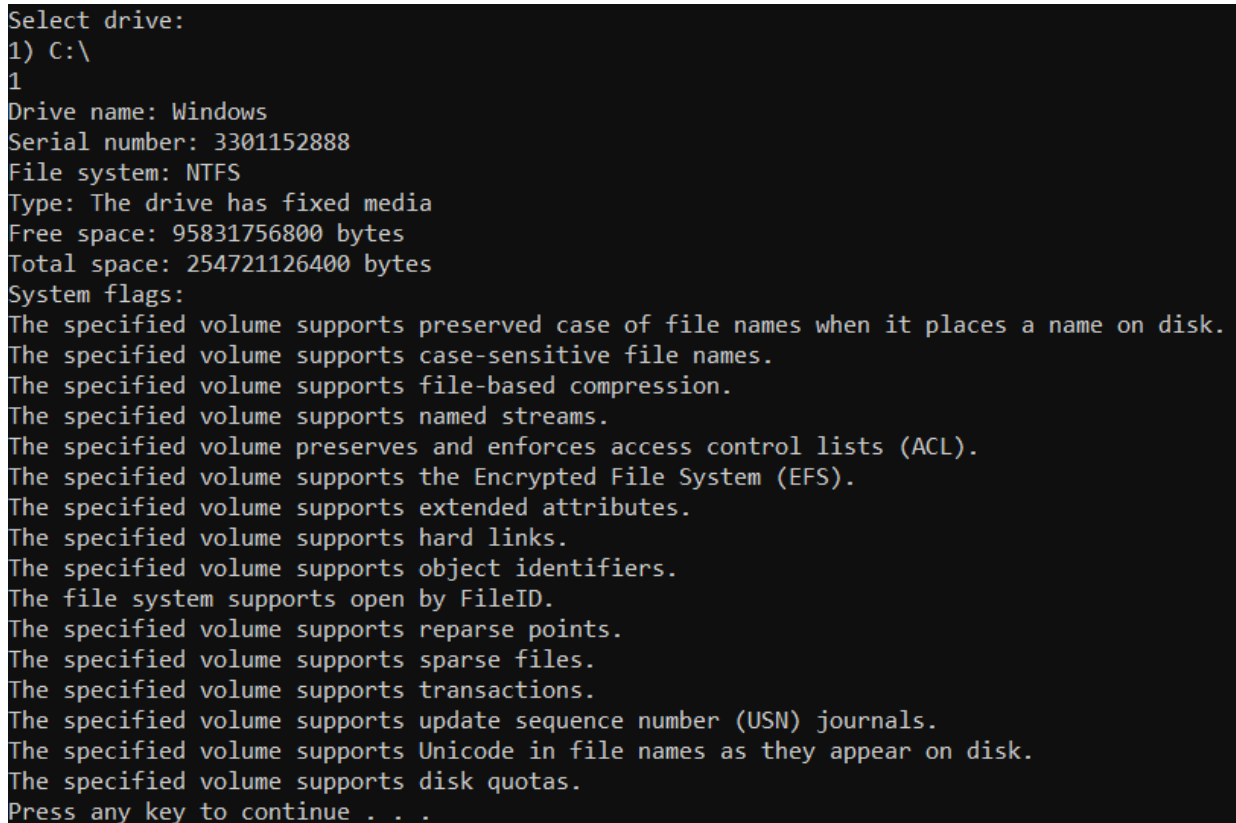
- Вывод списка доступных дисков:



```
Drive list:  
C:\  
Press any key to continue . . .
```

*Рисунок 1. Список дисков*

- Вывод информации о выбранном диске:



```
Select drive:  
1) C:\  
1  
Drive name: Windows  
Serial number: 3301152888  
File system: NTFS  
Type: The drive has fixed media  
Free space: 95831756800 bytes  
Total space: 254721126400 bytes  
System flags:  
The specified volume supports preserved case of file names when it places a name on disk.  
The specified volume supports case-sensitive file names.  
The specified volume supports file-based compression.  
The specified volume supports named streams.  
The specified volume preserves and enforces access control lists (ACL).  
The specified volume supports the Encrypted File System (EFS).  
The specified volume supports extended attributes.  
The specified volume supports hard links.  
The specified volume supports object identifiers.  
The file system supports open by FileID.  
The specified volume supports reparse points.  
The specified volume supports sparse files.  
The specified volume supports transactions.  
The specified volume supports update sequence number (USN) journals.  
The specified volume supports Unicode in file names as they appear on disk.  
The specified volume supports disk quotas.  
Press any key to continue . . .
```

*Рисунок 2. Информация о диске*

- Создание новой директории:

```
Enter the path of the new directory:
C:\Users\paloc\C++\OS\1\Новая директория
Directory created!
Press any key to continue . . .
```

*Рисунок 3. Создание новой директории*

- Удаление директории:

```
Enter the path to the directory to be deleted:
C:\Users\paloc\C++\OS\1\Новая директория
Directory was deleted!
Press any key to continue . . .
```

*Рисунок 4. Удаление директории*

- Создание файла:

```
Enter the path to the new file:
C:\Users\paloc\C++\OS\Для файлов\новый файл.txt
File created!
Press any key to continue . . .
```

*Рисунок 5. Создание файла*

- Копирование файла:

```
Enter the path to the file to copy:
C:\Users\paloc\C++\OS\Для файлов\новый файл.txt
Enter destination directory:
C:\Users\paloc\C++\OS\1\новый файл.txt
File copied!
Press any key to continue . . .
```

*Рисунок 6. Копирование файла*

- Перемещение файла:

```
Enter the path to the file to move:
C:\Users\paloc\C++\OS\1\новый файл.txt
Enter destination directory:
C:\Users\paloc\C++\OS\1\1_1\новый файл.txt
File moved!
Press any key to continue . . .
```

*Рисунок 7. Перемещение файла*

- Вывод атрибутов файла:

```
Enter the path to the file:
C:\Users\paloc\C++\OS\1\1_1\новый файл.txt
File have these attributes:
Archive
File created at this time:
Year: 2020
Month: 10
Day: 16
Time: 16:30
Press any key to continue . . .
```

*Рисунок 8. Атрибуты файла*

- Изменение атрибутов файла:

```
1 - Archive: 0
2 - Hidden: 0
3 - Normal: 1
4 - Not content indexed: 0
5 - Read only: 0
6 - System: 0
7 - Temporary: 0
0 - Save and exit
0
File attributes changed!
Press any key to continue . . .
```

*Рисунок 9. Изменение атрибутов файла*

- Изменение времени создания файла:

```
Enter the path to the file:
C:\Users\paloc\C++\OS\1\1_1\новый файл.txt
New file time is set!
Press any key to continue . . .
```

*Рисунок 10. Изменение времени создания файла*

### **3 Вывод для задания 1**

При выполнении задания 1 были изучены разные функции WinAPI. С помощью этих функций была получена информация о файловой системе, были протестированы различные возможности работы с файлами и директориями.

## 4 Задание 2

- Копирование файла различными блоками, кратными размеру кластера:

```
Enter the multiplicity:  
1  
The copy time: 16374 ms
```

```
Enter the multiplicity:  
2  
The copy time: 9172 ms
```

```
Enter the multiplicity:  
3  
The copy time: 6601 ms
```

```
Enter the multiplicity:  
4  
The copy time: 4942 ms
```

```
Enter the multiplicity:  
5  
The copy time: 4851 ms
```

```
Enter the multiplicity:  
6  
The copy time: 4026 ms
```

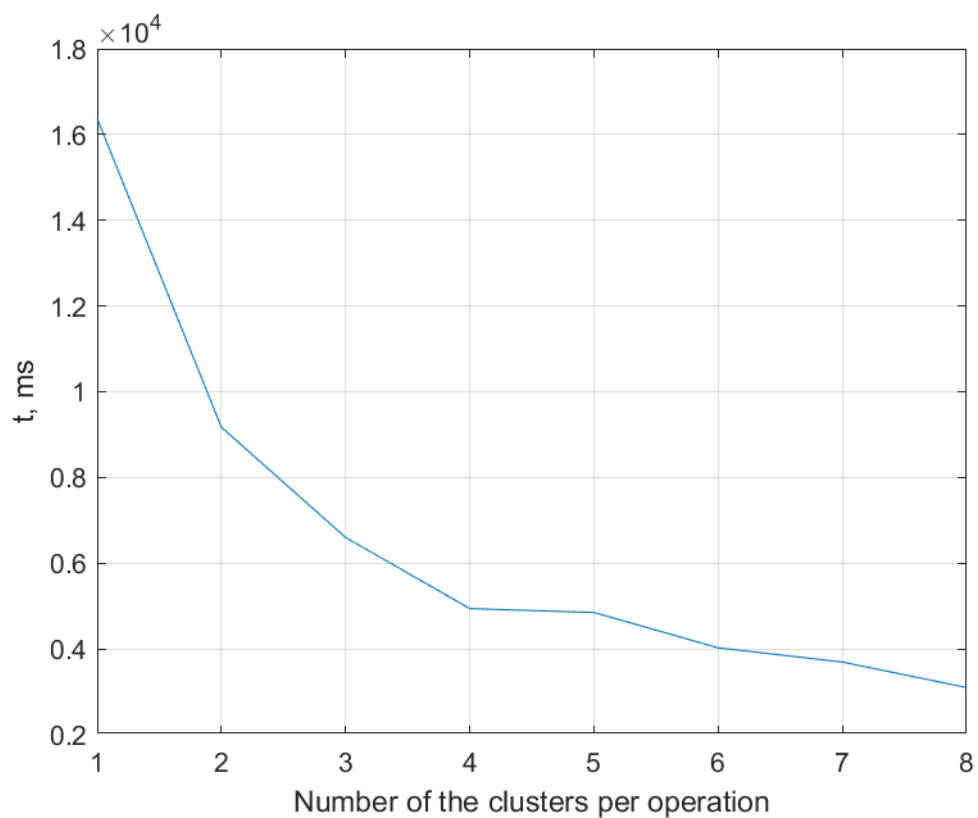
```
Enter the multiplicity:  
7  
The copy time: 3695 ms
```

```
Enter the multiplicity:  
8  
The copy time: 3100 ms
```

Проверим итоговый файл с помощью команды FC:

```
C:\Users\paloc\C++\OS\1\1_2>fc test.mkv result.mkv  
Comparing files test.mkv and RESULT.MKV  
FC: no differences encountered
```

Построим график:



По графику видно, что наибольший выигрыш времени можно получить только в самом начале. Потом время начинает убывать намного медленнее.

- Копирование файла с помощью различного числа перекрывающихся операций ввода вывода:

```
Enter the number of operation:
1
The copy time: 1016 ms
```

```
Enter the number of operation:
2
The copy time: 721 ms
```

```
Enter the number of operation:
4
The copy time: 620 ms
```

```
Enter the number of operation:
8
The copy time: 589 ms
```

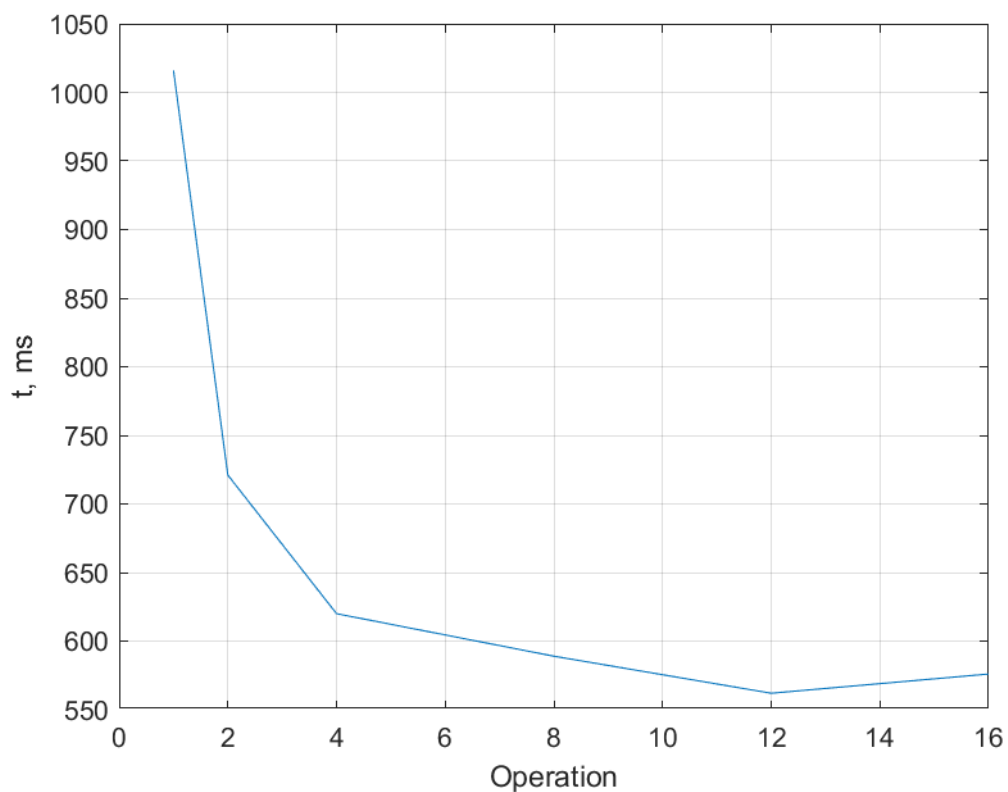
```
Enter the number of operation:
12
The copy time: 562 ms
```

```
Enter the number of operation:
16
The copy time: 576 ms
```

Проверим итоговый файл:

```
C:\Users\paloc\C++\05\1\1_2>fc test.mkv result.mkv
Comparing files test.mkv and RESULT.MKV
FC: no differences encountered
```

Построим график:



Из графика видно, что скорость копирования уменьшается с ростом числа операций. Для 16 операций наблюдается небольшое увеличение времени. Возможно это происходит из-за попадания на более приоритетный процесс.

## **5 Вывод для задания 2**

Во второй части задания мы научились копировать файлы при помощи асинхронных операций ввода-вывода. Из результатов следует, что асинхронный ввод-вывод может значительно уменьшить время обработки данных. Однако будет существовать предел, к которому будет стремиться время ввода-вывода.



## 6 Исходный код

### Задание 1

```
#include <iostream>
#include <map>
#include <cstdlib>
#include <limits>
#include <io.h>
#include <fcntl.h>
#include <windows.h>
```

```
DWORD const BUFFER_LENGTH = 100;
int const COUNT_OF_FLAG = 8;
```

```
void printWSTR(LPWSTR str, DWORD stringLength)
{
    for (unsigned int i = 0; i < stringLength; i++) {
        std::wcout << str[i];
    }
    std::wcout << std::endl;
}
```

```
void printWSTR(LPWSTR str)
{
    int i = 0;
    while (str[i] != '\0') {
        std::wcout << str[i];
        i++;
    }
    std::wcout << std::endl;
}
```

```
LPWSTR* parsingDrivesBuffer(LPWSTR drivesBuffer, DWORD bufferLength,
int& numberOfDrives)
{
    numberOfDrives = bufferLength / 4;
    LPWSTR* drives = new LPWSTR[numberOfDrives];
    for (int i = 0; i < numberOfDrives; i++) {
        drives[i] = new WCHAR[4];
        for (int j = 0; j < 4; j++) {
            drives[i][j] = drivesBuffer[4*i + j];
        }
    }
}
```

```

    return drives;
}

DWORD returnNewAttributes(DWORD oldAttributes)
{
    auto checkAttribute = [oldAttributes](DWORD _attributesFlag) -> bool {return
oldAttributes & _attributesFlag;};
    std::map<DWORD, bool> flags{{FILE_ATTRIBUTE_ARCHIVE,
checkAttribute(FILE_ATTRIBUTE_ARCHIVE)},
{FILE_ATTRIBUTE_HIDDEN,
checkAttribute(FILE_ATTRIBUTE_HIDDEN)},
{FILE_ATTRIBUTE_NORMAL,
checkAttribute(FILE_ATTRIBUTE_NORMAL)},
{FILE_ATTRIBUTE_NOT_CONTENT_INDEXED,
checkAttribute(FILE_ATTRIBUTE_NOT_CONTENT_INDEXED)},
{FILE_ATTRIBUTE_READONLY,
checkAttribute(FILE_ATTRIBUTE_READONLY)},
{FILE_ATTRIBUTE_SYSTEM,
checkAttribute(FILE_ATTRIBUTE_SYSTEM)},
{FILE_ATTRIBUTE_TEMPORARY,
checkAttribute(FILE_ATTRIBUTE_TEMPORARY)}};
    int userChoice;
    DWORD result = 0;
    std::map<DWORD, bool>::iterator i;
    do {
        system("cls");
        std::wcout << "1 - Archive: " << flags[FILE_ATTRIBUTE_ARCHIVE] <<
std::endl;
        std::wcout << "2 - Hidden: " << flags[FILE_ATTRIBUTE_HIDDEN] <<
std::endl;
        std::wcout << "3 - Normal: " << flags[FILE_ATTRIBUTE_NORMAL] <<
std::endl;
        std::wcout << "4 - Not content indexed: " <<
flags[FILE_ATTRIBUTE_NOT_CONTENT_INDEXED] <<std::endl;
        std::wcout << "5 - Read only: " << flags[FILE_ATTRIBUTE_READONLY] <<
std::endl;
        std::wcout << "6 - System: " << flags[FILE_ATTRIBUTE_SYSTEM] <<
std::endl;
        std::wcout << "7 - Temporary: " << flags[FILE_ATTRIBUTE_TEMPORARY]
<< std::endl;
        std::wcout << "0 - Save and exit" << std::endl;
        std::wcin >> userChoice;
        switch(userChoice) {
            case 1:

```

```

        flags[FILE_ATTRIBUTE_ARCHIVE] = !
flags[FILE_ATTRIBUTE_ARCHIVE];
        flags[FILE_ATTRIBUTE_NORMAL] = false;
        break;
    case 2:
        flags[FILE_ATTRIBUTE_HIDDEN] = !
flags[FILE_ATTRIBUTE_HIDDEN];
        flags[FILE_ATTRIBUTE_NORMAL] = false;
        break;
    case 3:
        if (!flags[FILE_ATTRIBUTE_NORMAL])
            for (i = flags.begin(); i != flags.end(); i++)
                i->second = false;
        flags[FILE_ATTRIBUTE_NORMAL] = !
flags[FILE_ATTRIBUTE_NORMAL];
        break;
    case 4:
        flags[FILE_ATTRIBUTE_NOT_CONTENT_INDEXED] = !
flags[FILE_ATTRIBUTE_NOT_CONTENT_INDEXED];
        flags[FILE_ATTRIBUTE_NORMAL] = false;
        break;
    case 5:
        flags[FILE_ATTRIBUTE_READONLY] = !
flags[FILE_ATTRIBUTE_READONLY];
        flags[FILE_ATTRIBUTE_NORMAL] = false;
        break;
    case 6:
        flags[FILE_ATTRIBUTE_SYSTEM] = !
flags[FILE_ATTRIBUTE_SYSTEM];
        flags[FILE_ATTRIBUTE_NORMAL] = false;
        break;
    case 7:
        flags[FILE_ATTRIBUTE_TEMPORARY] = !
flags[FILE_ATTRIBUTE_TEMPORARY];
        flags[FILE_ATTRIBUTE_NORMAL] = false;
        break;
    default:
        break;
}
} while (userChoice!=0);
for (i = flags.begin(); i != flags.end(); i++)
    result = result | (i->first * static_cast<DWORD>(i->second));
return result;
}

```

```

int inputVariable(int bottomBorder, int topBorder)
{
    int variable;
    do {
        std::wcin >> variable;
        if ((variable > topBorder) && (variable < bottomBorder)) {
            std::wcout << "Incorrect choice!" << std::endl;
            system("pause");
        }
    } while ((variable > topBorder) && (variable < bottomBorder));
    return variable;
}

void printFileAttributes(DWORD fileAttributes)
{
    auto checkAttribute = [fileAttributes](DWORD _attributesFlag) -> bool {return
fileAttributes & _attributesFlag;};

    if (checkAttribute(FILE_ATTRIBUTE_ARCHIVE))
        std::wcout << "Archive" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_COMPRESSED))
        std::wcout << "Compressed" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_DIRECTORY))
        std::wcout << "Directory" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_ENCRYPTED))
        std::wcout << "Encrypted" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_HIDDEN))
        std::wcout << "Hidden" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_NORMAL))
        std::wcout << "Normal" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_NOT_CONTENT_INDEXED))
        std::wcout << "Not conten indexed" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_OFFLINE))
        std::wcout << "Offline" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_READONLY))
        std::wcout << "Read only" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_SYSTEM))
        std::wcout << "System" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_TEMPORARY))
        std::wcout << "Temporary" << std::endl;
    if (checkAttribute(FILE_ATTRIBUTE_SPARSE_FILE))
        std::wcout << "Sparse" << std::endl;
}

```

```

}

void printFileSystemFlags(DWORD fileSystemFlags)
{
    auto checkAttribute = [fileSystemFlags](DWORD _systemFlag) -> bool {return
fileSystemFlags & _systemFlag;};

    if (checkAttribute(FILE_CASE_PRESERVED_NAMES))
        std::wcout << "The specified volume supports preserved case of file names
when it places a name on disk." << std::endl;
    if (checkAttribute(FILE_CASE_SENSITIVE_SEARCH))
        std::wcout << "The specified volume supports case-sensitive file names." <<
std::endl;
    if (checkAttribute(FILE_FILE_COMPRESSION))
        std::wcout << "The specified volume supports file-based compression." <<
std::endl;
    if (checkAttribute(FILE_NAMED_STREAMS))
        std::wcout << "The specified volume supports named streams." << std::endl;
    if (checkAttribute(FILE_PERSISTENT_ACLS))
        std::wcout << "The specified volume preserves and enforces access control lists
(ACL)." << std::endl;
    if (checkAttribute(FILE_READ_ONLY_VOLUME))
        std::wcout << "The specified volume is read-only." << std::endl;
    if (checkAttribute(FILE_SEQUENTIAL_WRITE_ONCE))
        std::wcout << "The specified volume supports a single sequential write." <<
std::endl;
    if (checkAttribute(FILE_SUPPORTS_ENCRYPTION))
        std::wcout << "The specified volume supports the Encrypted File System
(EFS)." << std::endl;
    if (checkAttribute(FILE_SUPPORTS_EXTENDED_ATTRIBUTES))
        std::wcout << "The specified volume supports extended attributes." << std::endl;
    if (checkAttribute(FILE_SUPPORTS_HARD_LINKS))
        std::wcout << "The specified volume supports hard links." << std::endl;
    if (checkAttribute(FILE_SUPPORTS_OBJECT_IDS))
        std::wcout << "The specified volume supports object identifiers." << std::endl;
    if (checkAttribute(FILE_SUPPORTS_OPEN_BY_FILE_ID))
        std::wcout << "The file system supports open by FileID." << std::endl;
    if (checkAttribute(FILE_SUPPORTS_REPARSE_POINTS))
        std::wcout << "The specified volume supports reparse points." << std::endl;
    if (checkAttribute(FILE_SUPPORTS_SPARSE_FILES))
        std::wcout << "The specified volume supports sparse files." << std::endl;
    if (checkAttribute(FILE_SUPPORTS_TRANSACTIONS))
        std::wcout << "The specified volume supports transactions." << std::endl;
    if (checkAttribute(FILE_SUPPORTS_USN_JOURNAL))

```

```

        std::wcout << "The specified volume supports update sequence number (USN)
journals." << std::endl;
        if (checkAttribute(FILE_UNICODE_ON_DISK))
            std::wcout << "The specified volume supports Unicode in file names as they
appear on disk." << std::endl;
        if (checkAttribute(FILE_VOLUME_IS_COMPRESSED))
            std::wcout << "The specified volume is a compressed volume, for example, a
DoubleSpace volume." << std::endl;
        if (checkAttribute(FILE_VOLUME_QUOTAS))
            std::wcout << "The specified volume supports disk quotas." << std::endl;
    }

```

```

void printDate(LPSYSTEMTIME time)
{
    std::wcout << "Year: " << time->wYear << std::endl;
    std::wcout << "Month: " << time->wMonth << std::endl;
    std::wcout << "Day: " << time->wDay << std::endl;
    std::wcout << "Time: " << time->wHour << ":" << time->wMinute << std::endl;
}

```

```

void printDriveType(UINT driveType)
{
    switch (driveType) {
        case 0:
            std::wcout << "The drive type cannot be determined";
            break;
        case 1:
            std::wcout << "The root path is invalid";
            break;
        case 2:
            std::wcout << "The drive has removable media";
            break;
        case 3:
            std::wcout << "The drive has fixed media";
            break;
        case 4:
            std::wcout << "The drive is a remote (network) drive";
            break;
        case 5:
            std::wcout << "The drive is a CD-ROM drive";
            break;
        case 6:
            std::wcout << "The drive is a RAM disk";
            break;
    }
}

```

```

    }
    std::wcout << std::endl;
}

void printError()
{
    std::wcout << "Error!" << std::endl;
    std::wcout << "Error code: " << GetLastError() << std::endl;
}

void printMenu()
{
    system("cls");
    std::wcout << "1 - Drive List" << std::endl;
    std::wcout << "2 - Drive Information" << std::endl;
    std::wcout << "3 - Create directory" << std::endl;
    std::wcout << "4 - Delete directory" << std::endl;
    std::wcout << "5 - Create file" << std::endl;
    std::wcout << "6 - Copy file" << std::endl;
    std::wcout << "7 - Move file" << std::endl;
    std::wcout << "8 - File attributes" << std::endl;
    std::wcout << "9 - Change file attributes" << std::endl;
    std::wcout << "10 - Change file creation time" << std::endl;
    std::wcout << "0 - Exit" << std::endl;
}

int main()
{
    int userChoice, numberOfDrives = 0, selectedDrive, i;
    std::wstring directoryOrFilePath, destinationDirectory;
    BOOL errorFlag;
    HANDLE fileHandle = NULL;
    UINT driveType;
    LPWSTR drivesBuffer, volumeNameBuffer, fileNameBuffer;
    LPWSTR* drives = nullptr;
    LPBY_HANDLE_FILE_INFORMATION fileInformation;
    LPSYSTEMTIME resultTime, currentTime;
    LPFILETIME fileTime;
    DWORD drivesBufferLength, fileAttributes;
    LPDWORD sectorsPerCluster, bytesPerSector, numberOfFreeCluster,
totalNumberOfCluster,
    volumeSerialNumber, maximumComponentLength, fileSystemFlags;

    sectorsPerCluster = new DWORD;

```

```

bytesPerSector = new DWORD;
numberOfFreeCluster = new DWORD;
totalNumberOfCluster = new DWORD;
volumeSerialNumber = new DWORD;
maximumComponentLength = new DWORD;
fileSystemFlags = new DWORD;

fileNameBuffer = new WCHAR[BUFFER_LENGTH];
volumeNameBuffer = new WCHAR[BUFFER_LENGTH];
drivesBuffer = new WCHAR[BUFFER_LENGTH];

fileInformation = new BY_HANDLE_FILE_INFORMATION;
fileTime = new FILETIME;
resultTime = new SYSTEMTIME;
currentTime = new SYSTEMTIME;

_setmode(_fileno(stdout), _O_U16TEXT);
_setmode(_fileno(stdin), _O_U16TEXT);
do {
    printMenu();
    std::wcin >> userChoice;
    std::wcin.get();
    std::wcin.get();
    system("cls");
    switch(userChoice) {
        case 1:
            drivesBufferLength = GetLogicalDriveStringsW(BUFFER_LENGTH,
drivesBuffer);
            if (drivesBufferLength == 0) {
                printError();
            }
            else {
                std::wcout << "Drive List:" << std::endl;
                printWSTR(drivesBuffer, drivesBufferLength);
                system("pause");
            }
            break;
        case 2:
            drivesBufferLength = GetLogicalDriveStringsW(BUFFER_LENGTH,
drivesBuffer);
            if (drivesBufferLength == 0) {
                printError();
            }
            else {

```



```

        for (i = 0; i < numberOfDrives; i++)
            delete[] drives[i];
        if (drives != nullptr)
            delete[] drives;
        drives = parsingDrivesBuffer(drivesBuffer, drivesBufferLength,
numberOfDrives);
        std::wcout << "Select drive:" << std::endl;
        for (i = 0; i < numberOfDrives; i++) {
            std::wcout << i + 1 << ") " ;
            printWSTR(drives[i], 4);
        }
        selectedDrive = inputVariable(1, numberOfDrives);
        errorFlag = GetDiskFreeSpaceW(drives[selectedDrive - 1],
sectorsPerCluster, bytesPerSector, numberOfFreeCluster, totalNumberOfCluster);
        if (!errorFlag)
            printError();
        else {
            driveType = GetDriveTypeW(drives[selectedDrive - 1]);
            errorFlag = GetVolumeInformationW(drives[selectedDrive - 1],
volumeNameBuffer, BUFFER_LENGTH, volumeSerialNumber,
maximumComponentLength, fileSystemFlags, fileSystemNameBuffer,
BUFFER_LENGTH);
            if (!errorFlag) {
                printError();
            }
            else {
                std::wcout << "Drive name: ";
                printWSTR(volumeNameBuffer);
                std::wcout << "Serial number: " << *volumeSerialNumber <<
std::endl;

                std::wcout << "File system: ";
                printWSTR(fileSystemNameBuffer);
                std::wcout << "Type: ";
                printDriveType(driveType);
                std::wcout << "Free space: " << 1ULL * (*bytesPerSector) *
(*sectorsPerCluster) * (*numberOfFreeCluster)
<< " bytes" << std::endl;
                std::wcout << "Total space: " << 1ULL * (*bytesPerSector) *
(*sectorsPerCluster) * (*totalNumberOfCluster)
<< " bytes" << std::endl;
                std::wcout << "System flags:" << std::endl;
                printFileSystemFlags(*fileSystemFlags);
            }
        }
    }
}

```

```

        system("pause");
    }
    break;
case 3:
    std::wcout << "Enter the path of the new directory:" << std::endl;
    std::getline(std::wcin, directoryOrFilePath);
    errorFlag = CreateDirectoryW(directoryOrFilePath.c_str(), NULL);
    if (!errorFlag)
        printError();
    else
        std::wcout << "Directory created!" << std::endl;
    system("pause");
    break;
case 4:
    std::wcout << "Enter the path to the directory to be deleted:" << std::endl;
    std::getline(std::wcin, directoryOrFilePath);
    errorFlag = RemoveDirectoryW(directoryOrFilePath.c_str());
    if (!errorFlag)
        printError();
    else
        std::wcout << "Directory was deleted!" << std::endl;
    system("pause");
    break;
case 5:
    std::wcout << "Enter the path to the new file:" << std::endl;
    std::getline(std::wcin, directoryOrFilePath);
    fileHandle = CreateFileW(directoryOrFilePath.c_str(), GENERIC_READ |
GENERIC_WRITE,
0, NULL, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
    if (fileHandle == INVALID_HANDLE_VALUE)
        printError();
    else
    {
        std::wcout << "File created!" << std::endl;
        errorFlag = CloseHandle(fileHandle);
        if (!errorFlag)
            printError();
    }
    system("pause");
    break;
case 6:
    std::wcout << "Enter the path to the file to copy:" << std::endl;
    std::getline(std::wcin, directoryOrFilePath);
    std::wcin.get();

```

```

        std::wcout << "Enter destination directory:" << std::endl;
        std::getline(std::wcin, destinationDirectory);
        errorFlag = CopyFileW(directoryOrFilePath.c_str(),
destinationDirectory.c_str(), TRUE);
        if (!errorFlag)
            printError();
        else
            std::wcout << "File copied!" << std::endl;
        system("pause");
        break;
    case 7:
        std::wcout << "Enter the path to the file to move:" << std::endl;
        std::getline(std::wcin, directoryOrFilePath);
        std::wcin.get();
        std::wcout << "Enter destination directory:" << std::endl;
        std::getline(std::wcin, destinationDirectory);
        errorFlag = MoveFileW(directoryOrFilePath.c_str(),
destinationDirectory.c_str());
        if (!errorFlag)
            printError();
        else
            std::wcout << "File moved!" << std::endl;
        system("pause");
        break;
    case 8:
        std::wcout << "Enter the path to the file:" << std::endl;
        std::getline(std::wcin, directoryOrFilePath);
        fileHandle = CreateFileW(directoryOrFilePath.c_str(), GENERIC_WRITE,
0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (fileHandle == INVALID_HANDLE_VALUE) {
            printError();
        }
        else {
            errorFlag = GetFileInformationByHandle(fileHandle, fileInformation);
            if (!errorFlag)
                printError();
            else {
                errorFlag =
FileTimeToSystemTime(&fileInformation>ftCreationTime, resultTime);
                if (!errorFlag)
                    printError();
                else {
                    errorFlag =
SystemTimeToTzSpecificLocalTime(NULL,resultTime, currentTime);

```

```

        if (!errorFlag)
            printError();
        else {
            std::wcout << "File have these attributes:" << std::endl;
            printFileAttributes(fileInformation->dwFileAttributes);
            std::wcout << "File created at this time:" << std::endl;
            printDate(currentTime);
            errorFlag = CloseHandle(fileHandle);
            if (!errorFlag)
                printError();
        }
    }
}
}
system("pause");
break;
case 9:
    std::wcout << "Enter the path to the file:" << std::endl;
    std::getline(std::wcin, directoryOrFilePath);
    fileHandle = CreateFileW(directoryOrFilePath.c_str(), GENERIC_READ,
0, NULL, OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS,
NULL);
    if (fileHandle == INVALID_HANDLE_VALUE)
        printError();
    else {
        fileAttributes = GetFileAttributesW(directoryOrFilePath.c_str());
        if (fileAttributes == INVALID_FILE_ATTRIBUTES)
            printError();
        else {
            fileAttributes = returnNewAttributes(fileAttributes);
            errorFlag = SetFileAttributesW(directoryOrFilePath.c_str(),
fileAttributes);
            if (!errorFlag)
                printError();
            else {
                std::wcout << "File attributes changed!" << std::endl;
                errorFlag = CloseHandle(fileHandle);
                if (!errorFlag)
                    printError();
            }
        }
    }
}
system("pause");
break;

```

```

    case 10:
        std::wcout << "Enter the path to the file:" << std::endl;
        std::getline(std::wcin, directoryOrFilePath);
        fileHandle = CreateFileW(directoryOrFilePath.c_str(),
FILE_WRITE_ATTRIBUTES,
0, NULL, OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS, NULL);
        if (fileHandle == INVALID_HANDLE_VALUE)
            printError();
        else {
            GetSystemTime(currentTime);
            errorFlag = SystemTimeToFileTime(currentTime, fileTime);
            if (!errorFlag)
                printError();
            else {
                errorFlag = SetFileTime(fileHandle, fileTime, NULL, NULL);
                if (!errorFlag)
                    printError();
                else {
                    std::wcout << "New file time is set!" << std::endl;
                    errorFlag = CloseHandle(fileHandle);
                    if (!errorFlag)
                        printError();
                }
            }
        }
        system("pause");
        break;
    default:
        if (userChoice != 0) {
            std::wcout << "Incorrect choice!" << std::endl;
            system("pause");
        }
    }
} while (userChoice != 0);

delete sectorsPerCluster;
delete bytesPerSector;
delete numberOfFreeCluster;
delete totalNumberOfCluster;
delete volumeSerialNumber;
delete maximumComponentLength;
delete fileSystemFlags;
delete fileInformation;
delete resultTime;

```

```
delete currentTime;  
delete fileTime;  
  
delete[] fileNameBuffer;  
delete[] volumeNameBuffer;  
delete[] drivesBuffer;  
  
return 0;  
}
```

## Задание 2

```
#include <iostream>
#include <cstdlib>
#include <io.h>
#include <fcntl.h>
#include <Windows.h>

int numberOfReadOperationStarted = 0;
int numberOfWriteOperationStarted = 0;

void printError()
{
    std::wcout << "Error!" << std::endl;
    std::wcout << "Error code: " << GetLastError() << std::endl;
}

VOID CALLBACK finishReadOperation(DWORD dwErrorCode, DWORD
dwNumberOfBytesTransferred, LPOVERLAPPED lpOverLapped)
{
    numberOfReadOperationStarted--;
}

VOID CALLBACK finishWriteOperation(DWORD dwErrorCode, DWORD
dwNumberOfBytesTransferred, LPOVERLAPPED lpOverLapped)
{
    numberOfWriteOperationStarted--;
}

int main()
{
    int userChoice, numberOfOverlappingOperation, multiplicity, bytesPerOperation,
    numberOfSectors, i;
    uint64_t fileSize, currentOffset;
    char **buffer;
    BOOL errorFlag;
    std::wstring filePath, newFilePath;
    LPDWORD sectorsPerCluster, bytesPerSector, numberOfFreeCluster,
totalNumberOfCluster, fileSizeHigh, fileSizeLow;
    DWORD startTime, endTime;
    HANDLE fileHandle, newFileHandle;
    LONG distanceToMove;
    PLONG distanceToMoveHigh;
    LPOVERLAPPED overlappedStructures;
```

```

sectorsPerCluster = new DWORD;
bytesPerSector = new DWORD;
numberOfFreeCluster = new DWORD;
totalNumberOfCluster = new DWORD;
fileSizeHigh = new DWORD;
fileSizeLow = new DWORD;
distanceToMoveHigh = new LONG;

_setmode(_fileno(stdout), _O_U16TEXT);
_setmode(_fileno(stdin), _O_U16TEXT);

std::wcout << "Enter the path to the file:" << std::endl;
std::getline(std::wcin, filePath);
std::wcin.get();
std::wcout << "Enter the path to the new file:" << std::endl;
std::getline(std::wcin, newFilePath);

errorFlag = GetDiskFreeSpaceW(L"C:\\", sectorsPerCluster,
    bytesPerSector, numberOfFreeCluster, totalNumberOfCluster);
if (errorFlag) {
    fileHandle = CreateFileW(filePath.c_str(), GENERIC_READ,
        0, NULL, OPEN_EXISTING, FILE_FLAG_NO_BUFFERING |
FILE_FLAG_OVERLAPPED, NULL);
    if (fileHandle != INVALID_HANDLE_VALUE) {
        newFileHandle = CreateFileW(newFilePath.c_str(), GENERIC_WRITE,
            0, NULL, CREATE_ALWAYS, FILE_FLAG_NO_BUFFERING |
FILE_FLAG_OVERLAPPED, NULL);
        if (newFileHandle != INVALID_HANDLE_VALUE) {
            *fileSizeLow = GetFileSize(fileHandle, fileSizeHigh);
            if (*fileSizeLow != INVALID_FILE_SIZE) {
                std::wcout << "1 - Serial" << std::endl << "2 - Parallel" << std::endl;
                std::wcin >> userChoice;
                fileSize = (static_cast<uint64_t>(*fileSizeHigh) << 32) | (*fileSizeLow);
                if (userChoice == 1) {
                    std::wcout << "Enter the multiplicity:" << std::endl;
                    std::wcin >> multiplicity;
                    bytesPerOperation = multiplicity * (*sectorsPerCluster) *
(*bytesPerSector);
                    numberOfOverlappingOperation = fileSize / bytesPerOperation;
                    buffer = new char*[numberOfOverlappingOperation + 1];
                    currentOffset = 0;
                    i = 0;
                    overlappedStructures = new OVERLAPPED[1];

```



```

overlappedStructures->Offset = 0;
overlappedStructures->OffsetHigh = 0;
startTime = timeGetTime();
do {
    buffer[i] = new char[bytesPerOperation];
    numberOfReadOperationStarted++;
    errorFlag = ReadFileEx(fileHandle,
static_cast<LPVOID>(buffer[i]), bytesPerOperation, overlappedStructures,
finishReadOperation);
    if (errorFlag) {
        SleepEx(-1, TRUE);
        i++;
        currentOffset = currentOffset + bytesPerOperation;
        overlappedStructures->Offset = currentOffset & 0xFFFFFFFF;
        overlappedStructures->OffsetHigh = currentOffset >> 32;
    } else
        printError();
} while ((currentOffset < fileSize) && errorFlag);
overlappedStructures->Offset = 0;
overlappedStructures->OffsetHigh = 0;
currentOffset = 0;
i = 0;
do {
    numberOfWriteOperationStarted++;
    errorFlag = WriteFileEx(newFileHandle,
static_cast<LPVOID>(buffer[i]), bytesPerOperation, overlappedStructures,
finishWriteOperation);
    if (errorFlag) {
        SleepEx(-1, TRUE);
        currentOffset = currentOffset + bytesPerOperation;
        delete[] buffer[i];
        i++;
        overlappedStructures->Offset = currentOffset & 0xFFFFFFFF;
        overlappedStructures->OffsetHigh = currentOffset >> 32;
    } else
        printError();
} while ((currentOffset < fileSize) && errorFlag);
distanceToMove = fileSize & 0xFFFFFFFF;
*distanceToMoveHigh = fileSize >> 32;
SetFilePointer(newFileHandle, distanceToMove,
distanceToMoveHigh, FILE_BEGIN);
SetEndOfFile(newFileHandle);
endTime = timeGetTime();

```

```

        std::wcout << "The copy time: " << endTime - startTime << " ms"
<<std::endl;
        system("pause");
    }
    else {
        std::wcout << "Enter the number of operation:" << std::endl;
        std::wcin >> numberOfOverlappingOperation;
        numberOfSectors = fileSize / (*sectorsPerCluster) / (*bytesPerSector);
        bytesPerOperation = numberOfSectors /
numberOfOverlappingOperation * (*sectorsPerCluster) * (*bytesPerSector);
        buffer = new char*[numberOfOverlappingOperation + 1];
        overlappedStructures = new
OVERLAPPED[numberOfOverlappingOperation + 1];
        currentOffset = 0;
        for (i = 0; i < numberOfOverlappingOperation + 1; i++)
        {
            overlappedStructures[i].Offset = currentOffset & 0xFFFFFFFF;
            overlappedStructures[i].OffsetHigh = currentOffset >> 32;
            buffer[i] = new char[bytesPerOperation];
            currentOffset = currentOffset + bytesPerOperation;
        }
        startTime = timeGetTime();
        for (i = 0; (i < numberOfOverlappingOperation + 1) && errorFlag;
i++)
        {
            errorFlag = ReadFileEx(fileHandle,
static_cast<LPVOID>(buffer[i]),
bytesPerOperation, &overlappedStructures[i], finishReadOperation);
            numberOfReadOperationStarted++;
        }
        for (i = 0; (i < numberOfOverlappingOperation + 1) && errorFlag;
i++)
        {
            errorFlag = WriteFileEx(newFileHandle,
static_cast<LPVOID>(buffer[i]),
bytesPerOperation, &overlappedStructures[i], finishWriteOperation);
            numberOfWriteOperationStarted++;
        }
        while (numberOfWriteOperationStarted != 0)
            SleepEx(-1, TRUE);
        endTime = timeGetTime();
        for (i = 0; i < numberOfOverlappingOperation + 1; i++)
            delete[] buffer[i];
        distanceToMove = fileSize & 0xFFFFFFFF;
    }
}

```

```

        *distanceToMoveHigh = fileSize >> 32;
        SetFilePointer(newFileHandle, distanceToMove,
distanceToMoveHigh, FILE_BEGIN);
        SetEndOfFile(newFileHandle);
        endTime = timeGetTime();
        std::wcout << "The copy time: " << endTime - startTime << " ms"
<<std::endl;
        system("pause");
    }
    errorFlag = CloseHandle(fileHandle);
    errorFlag = CloseHandle(newFileHandle);
    if (!errorFlag)
        printError();
    delete[] buffer;
    delete[] overlappedStructures;
    }
    else
        printError();
    }
    else
        printError();
    }
    else
        printError();
    }
    else
        printError();

    delete sectorsPerCluster;
    delete bytesPerSector;
    delete numberOfFreeCluster;
    delete totalNumberOfCluster;
    delete fileSizeHigh;
    delete fileSizeLow;
    delete distanceToMoveHigh;
    return 0;
}

```