**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ВТ**

отчет

**По Лабораторной работе №3**

**По дисциплине «Операционные системы»**

Тема: Управление файловой системой

|  |  |  |
| --- | --- | --- |
| Студент гр. 7307 |  | Торопов В.А. |
| Преподаватель |  | Тимофеев А.В. |

Санкт-Петербург

2019

**Цель работы**

Исследовать структуры данных Windows, используемые для обеспечения безопасности.

**Основные теоретические положения**

Process Explorer – утилита, заменяющая Диспетчер задач Windows, отображает более подробную информацию о процессах и потоках, включая их родство, загруженные DLL и открытые дескрипторы объектов. Возможно получить SID записей.

PsGetSid ­­­­– утилита специально предназначена для получения SID разных учетных записей. Данная утилита входит в набор PsTools.

**Обработка экспериментальных данных**

**Задание 2.1.** Определить идентификатор защиты SID текущего пользователя.

Было разработано консольное приложение, поддерживающее несколько режимов работы. Запустим приложение, получим его PID = 14472= 388816.

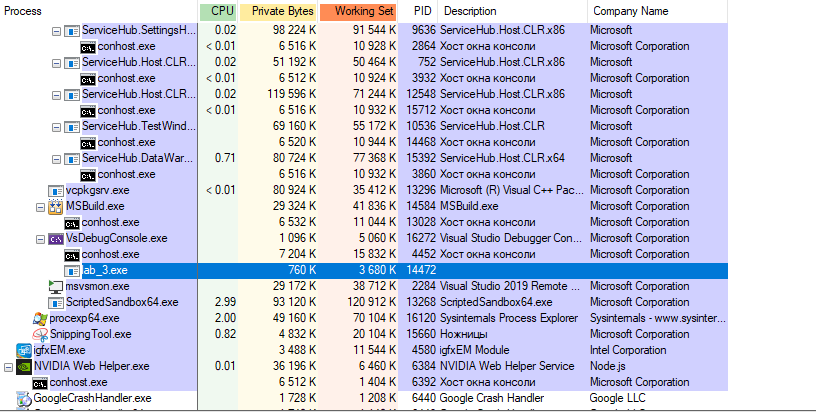


Рисунок 1. Процесс в ProcessExplorer

Последовательно выполним каждый из доступных пунктов и посмотрим количество используемых дескрипторов с помощью LiveKD.

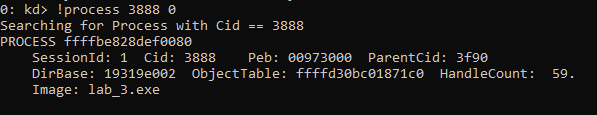


Рисунок 2. Вывод команды !process

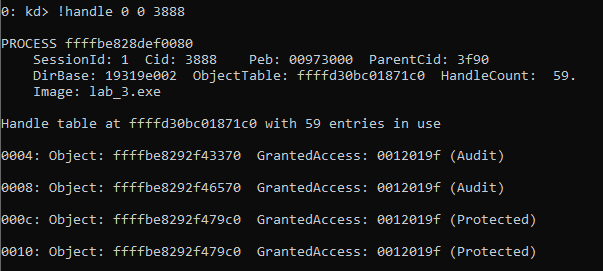


Рисунок 3. Вывод команды !handle

Протестируем работу консольного приложения и просмотрим количество используемых дескрипторов.

**1. Список доступных дисков**.

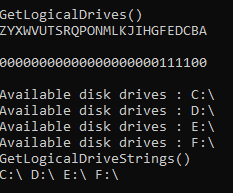


Рисунок 4. Результат выполнения команд вывода списка дисков

Количество открытых дескрипторов не изменилось.

**2. Вывод информации о диске**

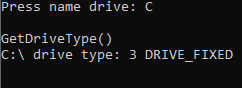


Рисунок 5. Результат выполнения команды GetDriveType

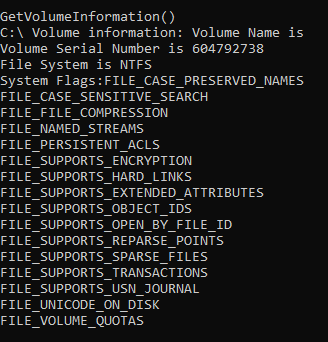


Рисунок 6. Результат выполнения команды GetVolumeInformation

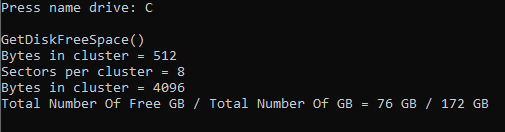


Рисунок 7. Результат выполнения команды GetDiskFreeSpace

Количество открытых дескрипторов во всех трех случаях не изменилось.

**3. Создание и удаление заданных каталогов**

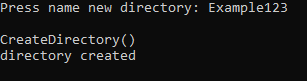


Рисунок 8. Результат создания дериктории

Количество открытых дескрипторов не изменилось

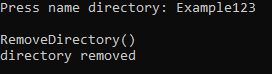


Рисунок 9. Результат удаления дериктории

Количество открытых дескрипторов не изменилось

**4. Создание файла**

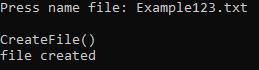


Рисунок 10. Результат выполнения команды Create File

Количество открытых дескрипторов не изменилось

**5. Копирование и перемещение файлов между каталогами**

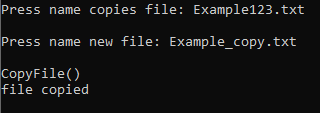


Рисунок 11. Результат выполнения команды CopyFile

Количество открытых дескрипторов увеличилось на 17, так как Windows использует внутренние объекты.

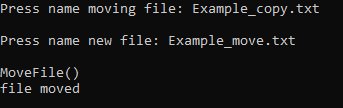


Рисунок 12. Результат выполнения команды MoveFile

Количество дескрипторов не изменилось.

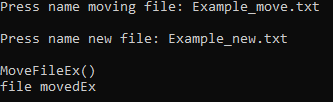


Рисунок 13. Результат выполнения команды MoveFileEx

Количество дескрипторов не изменилось.

**6. Анализ и изменение атрибутов файлов**

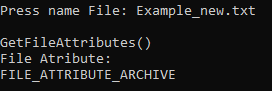


Рисунок 14. Результат выполнения команды GetFileAtributes

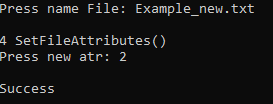


Рисунок 15. Результат выполнения команды SetFileAtributes

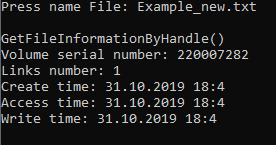


Рисунок 16. Результат выполнения команды GetFileInformationByHandle

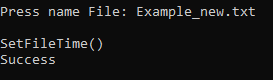


Рисунок 17. Результат выполнения команды SetFileTime

Количество открытых дескрипторов во всех трех случаях не изменилось.

Проверим работу функции GetFileTime() с закрытым дескриптором.

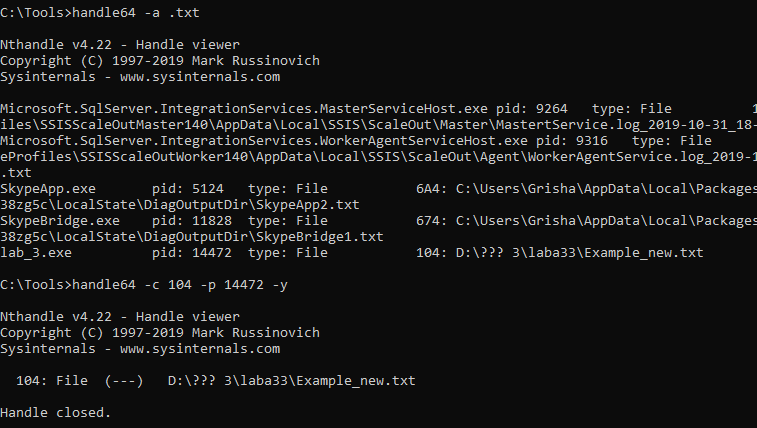


Рисунок 18. Ручное закрытие дескриптора файла

После открытия файла, программа останавливается, после закрытия дескриптора файла программа не может прочитать данные.

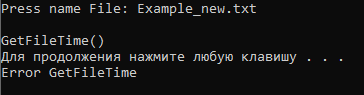


Рисунок 19. Завершение выполнения команды с ошибкой

Вывод: Важно правильно обрабатывать ошибки, связанные с дескрипторами объектов, это позволит избежать экстренных завершений программы. Использование CloseHandle позволяет избежать утечки дескрипторов при работе с файлами. Иногда Windows использует дополнительные дескрипторы разных типов.

**Задание 3.2.** Копирование файла с помощью операция перекрывающегося ввода-

вывода.

Проведём эксперимент для нахождения оптимального количества перекрывающих операций и оптимального размера блока чтения/записи. Размер кластера 4096 байт. Для проведения временных замеров воспользуемся файлом со случайным содержимым, размером 405 778 кб.

Первый эксперимент с разным размером обрабатываемого блока.

*Таблица 1. Время выполнения в зависимости размера блока*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Множитель | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Время, мс | 18291 | 11881 | 10521 | 8489 | 8153 | 8205 | 8154 | 8571 | 8351 | 11887 |

Рисунок 20. Результат handle до запуска копирования

Второй эксперимент зафиксируем размер блока равным размеру кластера (4096) и

зафиксируем время выполнения для разного числа перекрывающихся операций ввода и

вывода.

*Таблица 2. Время выполнения в зависимости от количества операций*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Число операций, шт | 1 | 2 | 4 | 8 | 12 | 16 |
| Время, мс | 11196 | 9197 | 8755 | 8197 | 8389 | 8422 |

Рисунок 22. Результат handle до запуска копирования

Вывод: исходя из результатов эксперимента, копирование файлов большими блоками

дает существенное ускорение до тех пор, пока увеличение множителя сокращает количество

требуемых операций чтения, в данном эксперименте максимальное ускорение наблюдалось при 5\* 4096 = 20480 байт, затем ускорение приостановилось. Увеличение же количества

операций замедлило работу, поэтому оптимальность выполнения наблюдалось при 8 операции.

**Выводы**

WinApi предоставляет программисту широкие возможности для взаимодействия с

операционной системой. Применение этих возможностей помогает оптимизировать вашу

программу.

Приложение 1

#include <windows.h>

#include <iostream>

#include <string>

using namespace std;

void showLogicalDrives() {

int n;

char dd[4];

DWORD dr = GetLogicalDrives();

int x = 1 << 25;

cout << "GetLogicalDrives()" << endl;

for (int i = 25; i >= 0; i--)

cout << char(65 + i);

cout << endl<<endl;

for (int i = 0; i < 26; i++, x >>= 1) {

cout << ((x & dr )?1:0);

}

cout << endl<< endl;

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

{

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

if (n == 1)

{

dd[0] = char(65 + i); dd[1] = ':'; dd[2] = '\\'; dd[3] = 0;

cout << "Available disk drives : " << dd << endl;

}

}

cout << "GetLogicalDriveStrings()" << endl;

DWORD length = GetLogicalDriveStrings(0, NULL);

TCHAR a[512];

GetLogicalDriveStrings(length, a);

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

wprintf(L"%c", a[i]);

}

}

void showDriveType() {

int d;

cout << "Press name drive: ";

string str;

cin >> str;

str = str + ":\\";

cout << endl;

d = GetDriveTypeA(str.c\_str());

cout << "GetDriveType()" << endl;

cout << str << " drive type: ";

printf("%d", d);

if (d == DRIVE\_UNKNOWN) cout << " DRIVE\_UNKNOWN" << endl;

if (d == DRIVE\_NO\_ROOT\_DIR) cout << " DRIVE NO ROOT DIR" << endl;

if (d == DRIVE\_REMOVABLE) cout << " DRIVE\_REMOVABLE" << endl;

if (d == DRIVE\_FIXED) cout << +" DRIVE\_FIXED" << endl;

if (d == DRIVE\_REMOTE) cout << " DRIVE\_REMOTE" << endl;

if (d == DRIVE\_CDROM) cout << " DRIVE\_CDROM" << endl;

if (d == DRIVE\_RAMDISK) cout << " DRIVE\_RAMDISK" << endl;

}

void showVolumeInformation() {

char VolumeNameBuffer[100];

char FileSystemNameBuffer[100];

unsigned long VolumeSerialNumber;

DWORD FileSystemFlags;

cout << "Press name drive: ";

string str;

cin >> str;

str = str + ":\\";

cout << endl;

cout << "GetVolumeInformation()" << endl;

cout << str << " Volume information: ";

BOOL GetVolumeInformationFlag = GetVolumeInformationA(

"C:\\",

VolumeNameBuffer,

100,

&VolumeSerialNumber,

NULL, //&MaximumComponentLength,

&FileSystemFlags, //&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;

cout << "System Flags:";

if (FileSystemFlags & FILE\_CASE\_PRESERVED\_NAMES) cout << "FILE\_CASE\_PRESERVED\_NAMES" << endl;

if (FileSystemFlags & FILE\_CASE\_SENSITIVE\_SEARCH) cout << "FILE\_CASE\_SENSITIVE\_SEARCH" << endl;

if (FileSystemFlags & 0x20000000) cout << "FILE\_DAX\_VOLUME" << endl;

if (FileSystemFlags & FILE\_FILE\_COMPRESSION) cout << +"FILE\_FILE\_COMPRESSION" << endl;

if (FileSystemFlags & FILE\_NAMED\_STREAMS) cout << "FILE\_NAMED\_STREAMS" << endl;

if (FileSystemFlags & FILE\_PERSISTENT\_ACLS) cout << "FILE\_PERSISTENT\_ACLS" << endl;

if (FileSystemFlags & FILE\_READ\_ONLY\_VOLUME) cout << "FILE\_READ\_ONLY\_VOLUME" << endl;

if (FileSystemFlags & FILE\_SEQUENTIAL\_WRITE\_ONCE) cout << "FILE\_SEQUENTIAL\_WRITE\_ONCE" << endl;

if (FileSystemFlags & FILE\_SUPPORTS\_ENCRYPTION) cout << "FILE\_SUPPORTS\_ENCRYPTION" << endl;

if (FileSystemFlags & FILE\_SUPPORTS\_HARD\_LINKS) cout << "FILE\_SUPPORTS\_HARD\_LINKS" << endl;

if (FileSystemFlags & FILE\_SUPPORTS\_EXTENDED\_ATTRIBUTES) cout << "FILE\_SUPPORTS\_EXTENDED\_ATTRIBUTES" << endl;

if (FileSystemFlags & FILE\_SUPPORTS\_OBJECT\_IDS) cout << "FILE\_SUPPORTS\_OBJECT\_IDS" << endl;

if (FileSystemFlags & FILE\_SUPPORTS\_OPEN\_BY\_FILE\_ID) cout << "FILE\_SUPPORTS\_OPEN\_BY\_FILE\_ID" << endl;

if (FileSystemFlags & FILE\_SUPPORTS\_REPARSE\_POINTS) cout << "FILE\_SUPPORTS\_REPARSE\_POINTS" << endl;

if (FileSystemFlags & FILE\_SUPPORTS\_SPARSE\_FILES) cout << "FILE\_SUPPORTS\_SPARSE\_FILES" << endl;

if (FileSystemFlags & FILE\_SUPPORTS\_TRANSACTIONS) cout << "FILE\_SUPPORTS\_TRANSACTIONS" << endl;

if (FileSystemFlags & FILE\_SUPPORTS\_USN\_JOURNAL) cout << "FILE\_SUPPORTS\_USN\_JOURNAL" << endl;

if (FileSystemFlags & FILE\_UNICODE\_ON\_DISK) cout << "FILE\_UNICODE\_ON\_DISK" << endl;

if (FileSystemFlags & FILE\_VOLUME\_IS\_COMPRESSED) cout << "FILE\_VOLUME\_IS\_COMPRESSED" << endl;

if (FileSystemFlags & FILE\_VOLUME\_QUOTAS) cout << "FILE\_VOLUME\_QUOTAS" << endl;

}

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

}

void showDiskFreeSpace() {

/\*lpSectorsPerCluster,

lpBytesPerSector,

lpNumberOfFreeClusters,

lpTotalNumberOfClusters\*/

DWORD lpSectorsPerCluster,

lpBytesPerSector,

lpNumberOfFreeClusters,

lpTotalNumberOfClusters;

cout << "Press name drive: ";

string str;

cin >> str;

cout << endl;

cout << "GetDiskFreeSpace()"<<endl;

BOOL GetDiskFreeSpaceFlag = GetDiskFreeSpaceA(

(str + ":\\").c\_str(), // directory name

&lpSectorsPerCluster,

&lpBytesPerSector,

&lpNumberOfFreeClusters,

&lpTotalNumberOfClusters

);

if (GetDiskFreeSpaceFlag != 0)

{

cout << "Bytes in sector = " << (unsigned long)lpBytesPerSector << endl;

cout << "Sectors per cluster = " << (unsigned long)lpSectorsPerCluster << endl;

cout << "Bytes in cluster = " << (unsigned long)lpSectorsPerCluster \* lpBytesPerSector << endl;

double lpn = double((unsigned long)(lpNumberOfFreeClusters)/1024/1024);

cout << "Total Number Of Free GB / Total Number Of GB = " << lpn \* lpBytesPerSector \* lpSectorsPerCluster / 1024 << " GB" << " / "

<< (double) (lpTotalNumberOfClusters /1024 / 1024)\*lpBytesPerSector \* lpSectorsPerCluster / 1024

<< " GB" << endl;

}

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

}

void menuDrive() {

int x;

cout << "1 GetDriveType" << endl;

cout << "2 GetVolumeInformation" << endl;

cout << "3 GetDiskFreeSpace" << endl;

cin >> x;

switch (x) {

case 1:

showDriveType();

break;

case 2:

showVolumeInformation();

break;

case 3:

showDiskFreeSpace();

break;

default:

cout << "NOTHING!!!";

}

}

void menuDirectory() {

int x;

string str;

cout << "1 CreateDirectory" << endl;

cout << "2 RemoveDirectory" << endl;

cin >> x;

switch (x) {

case 1:

cout << "Press name new directory: ";

cin >> str;

cout << endl;

cout << "CreateDirectory()"<< endl;

if (CreateDirectoryA(str.c\_str(), NULL))

cout << "directory created" << endl;

else

cout << "error create directory" << endl;

break;

case 2:

cout << "Press name directory: ";

cin >> str;

cout << endl;

cout << "RemoveDirectory()" << endl;;

if (RemoveDirectoryA(str.c\_str()))

cout << "directory removed" << endl;

else

cout << "error remove directory" << endl;

break;

default:

cout << "NOTHING!!!";

}

}

void menuFile() {

int x;

string str, toStr;

wstring q;

HANDLE handle;

cout << "1 CreateFile " << endl;

cout << "2 CopyFile" << endl;

cout << "3 MoveFile" << endl;

cout << "4 MoveFileEx" << endl;

cin >> x;

switch (x) {

case 1:

cout << "Press name file: ";

cin >> str;

cout << endl;

cout << "CreateFile()" << endl;

handle = CreateFileA(str.c\_str(), GENERIC\_READ | GENERIC\_WRITE, 0, NULL, CREATE\_NEW, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (handle == INVALID\_HANDLE\_VALUE) {

printf("ERROR !!%x \n", GetLastError());

}

else

cout << "file created" << endl;

CloseHandle(handle);

break;

case 2:

cout << "Press name copies file: ";

cin >> str;

cout << "\nPress name new file: ";

cin >> toStr;

cout << endl;

cout << "CopyFile()" << endl;

if (CopyFileA(str.c\_str(), toStr.c\_str(), FALSE))

cout << "file copied" << endl;

else

cout << "error file copy" << endl;

break;

case 3:

cout << "Press name moving file: ";

cin >> str;

cout << "\nPress name new file: ";

cin >> toStr;

cout << endl;

cout << "MoveFile()" << endl;

if (MoveFileA(str.c\_str(), toStr.c\_str()))

cout << "file moved" << endl;

else

cout << "error file move" << endl;

break;

case 4:

cout << "Press name moving file: ";

cin >> str;

cout << "\nPress name new file: ";

cin >> toStr;

cout << endl;

cout << "MoveFileEx()" << endl;

if (MoveFileExA(str.c\_str(), toStr.c\_str(), MOVEFILE\_REPLACE\_EXISTING))

cout << "file movedEx" << endl;

else

cout << "error file moveEx" << endl;

break;

default:

cout << "NOTHING!!!";

}

}

void menuAtribute() {

HANDLE handle;

int x;

DWORD atribute;

string str;

cout << "1 GetFileAttributes " << endl;

cout << "2 GetFileInformationByHandle" << endl;

cout << "3 GetFileTime" << endl;

cout << "4 SetFileAttributes" << endl;

cout << "5 SetFileTime" << endl;

cin >> x;

switch (x) {

case 1:

cout << "Press name File: ";

cin >> str;

cout << endl;

cout << "GetFileAttributes()" << endl;

atribute = (GetFileAttributesA(str.c\_str()));

if (atribute != INVALID\_FILE\_ATTRIBUTES) {

cout << "File Atribute: " << endl;

if (atribute & FILE\_ATTRIBUTE\_ARCHIVE) cout << "FILE\_ATTRIBUTE\_ARCHIVE" << endl;

if (atribute & FILE\_ATTRIBUTE\_COMPRESSED) cout << "FILE\_ATTRIBUTE\_COMPRESSED" << endl;

if (atribute & FILE\_ATTRIBUTE\_DEVICE) cout << "FILE\_ATTRIBUTE\_DEVICE" << endl;

if (atribute & FILE\_ATTRIBUTE\_DIRECTORY) cout << +"FILE\_ATTRIBUTE\_DIRECTORY" << endl;

if (atribute & FILE\_ATTRIBUTE\_ENCRYPTED) cout << "FILE\_ATTRIBUTE\_ENCRYPTED" << endl;

if (atribute & FILE\_ATTRIBUTE\_HIDDEN) cout << "FILE\_ATTRIBUTE\_HIDDEN" << endl;

if (atribute & 0x8000) cout << "FILE\_ATTRIBUTE\_INTEGRITY\_STREAM" << endl;

if (atribute & FILE\_ATTRIBUTE\_NORMAL) cout << "FILE\_ATTRIBUTE\_NORMAL" << endl;

if (atribute & FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED) cout << "FILE\_ATTRIBUTE\_NOT\_CONTENT\_INDEXED" << endl;

if (atribute & 0x20000) cout << "FILE\_ATTRIBUTE\_NO\_SCRUB\_DATA" << endl;

if (atribute & FILE\_ATTRIBUTE\_OFFLINE) cout << "FILE\_ATTRIBUTE\_OFFLINE" << endl;

if (atribute & FILE\_ATTRIBUTE\_READONLY) cout << "FILE\_ATTRIBUTE\_READONLY" << endl;

if (atribute & 0x400000) cout << "FILE\_ATTRIBUTE\_RECALL\_ON\_DATA\_ACCESS" << endl;

if (atribute & 0x40000) cout << "FILE\_ATTRIBUTE\_RECALL\_ON\_OPEN" << endl;

if (atribute & FILE\_ATTRIBUTE\_REPARSE\_POINT) cout << "FILE\_ATTRIBUTE\_REPARSE\_POINT" << endl;

if (atribute & FILE\_ATTRIBUTE\_SPARSE\_FILE) cout << "FILE\_ATTRIBUTE\_SPARSE\_FILE" << endl;

if (atribute & FILE\_ATTRIBUTE\_SYSTEM) cout << "FILE\_ATTRIBUTE\_SYSTEM" << endl;

if (atribute & FILE\_ATTRIBUTE\_TEMPORARY) cout << "FILE\_ATTRIBUTE\_TEMPORARY" << endl;

if (atribute & FILE\_ATTRIBUTE\_VIRTUAL) cout << "FILE\_ATTRIBUTE\_VIRTUAL" << endl;

}

else {

cout << "Wrong path or file for a network share" << endl;

}

break;

case 2:

{

cout << "Press name File: ";

cin >> str;

cout << endl;

cout << "GetFileInformationByHandle()" << endl;

handle = CreateFileA(str.c\_str(), GENERIC\_READ | GENERIC\_WRITE, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (handle != INVALID\_HANDLE\_VALUE) {

BY\_HANDLE\_FILE\_INFORMATION fileInfo;

boolean f = GetFileInformationByHandle(handle, &fileInfo);

if (f) {

SYSTEMTIME stime;

cout << "Volume serial number: " << (unsigned long)fileInfo.dwVolumeSerialNumber << endl;

cout << "Links number: " << (unsigned long)fileInfo.nNumberOfLinks << endl;

FileTimeToSystemTime(&(fileInfo.ftCreationTime), &stime);

printf("Create time: %d.%d.%d %d:%d\n", stime.wDay, stime.wMonth, stime.wYear, stime.wHour, stime.wMinute);

FileTimeToSystemTime(&(fileInfo.ftLastAccessTime), &stime);

printf("Access time: %d.%d.%d %d:%d\n", stime.wDay, stime.wMonth, stime.wYear, stime.wHour, stime.wMinute);

FileTimeToSystemTime(&(fileInfo.ftLastWriteTime), &stime);

printf("Write time: %d.%d.%d %d:%d\n", stime.wDay, stime.wMonth, stime.wYear, stime.wHour, stime.wMinute);

}

else

cout << "error GetFileInformationByHandle" << endl;

}

else

cout << "INVALID\_HANDLE\_VALUE";

CloseHandle(handle); }

break;

case 3: {

cout << "Press name File: ";

cin >> str;

cout << endl;

cout << "GetFileTime()" << endl;

handle = CreateFileA(str.c\_str(), GENERIC\_READ | GENERIC\_WRITE, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (handle != INVALID\_HANDLE\_VALUE) {

SYSTEMTIME stime;

FILETIME crtime, actime, wrtime;

system("pause");

if (GetFileTime(handle, &crtime, &actime, &wrtime)) {

FileTimeToSystemTime(&crtime, &stime);

printf("Create time: %d.%d.%d %d:%d\n", stime.wDay, stime.wMonth, stime.wYear, stime.wHour, stime.wMinute);

FileTimeToSystemTime(&actime, &stime);

printf("Access time: %d.%d.%d %d:%d\n", stime.wDay, stime.wMonth, stime.wYear, stime.wHour, stime.wMinute);

FileTimeToSystemTime(&wrtime, &stime);

printf("Write time: %d.%d.%d %d:%d\n", stime.wDay, stime.wMonth, stime.wYear, stime.wHour, stime.wMinute);

CloseHandle(handle);

}

else {

cout << "Error GetFileTime";

}

}

else

cout << "INVALID\_HANDLE\_VALUE";

}

break;

case 4:

cout << "Press name File: ";

cin >> str;

cout << "\n4 SetFileAttributes()" << endl;

int newatribute;

cout << "Press new atr: ";

cin >>hex>> newatribute;

cin >> dec;

cout << endl;

if (SetFileAttributesA(str.c\_str(), newatribute))

cout << "Success";

else

cout << "Error SetFileAttribute";

break;

case 5: {

cout << "Press name File: ";

cin >> str;

cout << endl;

cout << "SetFileTime()" << endl;

handle = CreateFileA(str.c\_str(), GENERIC\_READ | GENERIC\_WRITE, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (handle != INVALID\_HANDLE\_VALUE) {

SYSTEMTIME stime;

FILETIME wrtime;

GetSystemTime(&stime);

SystemTimeToFileTime(&stime, &wrtime);

if (SetFileTime(handle, (LPFILETIME)NULL, (LPFILETIME)NULL, &wrtime))

cout << "Success";

else

cout << "Error SetFileTime";

CloseHandle(handle);

}

else

cout << "INVALID\_HANDLE\_VALUE";

break;

}

default:

printf("ERROR %x \n", GetLastError());

cout << "NOTHING!!!";

}

}

int main()

{

int x = -1;

while (x != 0) {

cout << "1 showLogicalDrives " << endl;

cout << "2 menuDrive" << endl;

cout << "3 menuDirectory" << endl;

cout << "4 menuFile" << endl;

cout << "5 menuAtribute" << endl;

cout << "0 Exit" << endl;

cout << "Press number: ";

cin >> x;

cout << endl;

switch (x) {

case 1:

showLogicalDrives();

break;

case 2:

menuDrive();

break;

case 3:

menuDirectory();

break;

case 4:

menuFile();

break;

case 5:

menuAtribute();

break;

}

cout << endl << endl;

}

}

Приложение 2

#include <windows.h>

#include <winbase.h>

#include <iostream>

#include <fstream>

#include <vector>

#include <chrono>

using namespace std;

long long write\_time = 0;

char\*\* info;

DWORD global\_i = 0;

int mul;

DWORD SizeCluster;

int operCount;

VOID WINAPI FileWrittenCallback(DWORD dwErrorCode, DWORD dwBytesTransferred, LPOVERLAPPED lpOverlapped) {

if (dwErrorCode != 0) {

cout << "Wrong write to file! Error:" << dwErrorCode << ", AddrOverlapped: " << lpOverlapped << endl;

}

else {

//cout << "Good Writing: Transferred: " << dwBytesTransferred<<" , AddrOverlapped: " <<lpOverlapped<< endl;

}

//cout << "Good Writing"<<endl;\*/

write\_time++;

}

VOID WINAPI CopyFinal(DWORD dwErrorCode, DWORD dwBytesTransferred, LPOVERLAPPED lpOverlapped) {

if (dwErrorCode != 0) {

cout << "CompletionRoutine: Unable to write to file! Error:" << dwErrorCode << ", AddrOverlapped: " << lpOverlapped << endl;

}

else {

OVERLAPPED overlapped;

overlapped.Offset = lpOverlapped->Offset;

overlapped.OffsetHigh = lpOverlapped->OffsetHigh;

DWORD t = WriteFileEx(lpOverlapped->hEvent, info[global\_i], SizeCluster, &overlapped,

(LPOVERLAPPED\_COMPLETION\_ROUTINE)FileWrittenCallback);

global\_i++;

}

}

int BlockCopying() {

HANDLE handleRead = CreateFileA("testfile.txt", GENERIC\_READ, FILE\_SHARE\_READ, nullptr, OPEN\_EXISTING, FILE\_FLAG\_NO\_BUFFERING | FILE\_FLAG\_OVERLAPPED , nullptr);

if (handleRead == INVALID\_HANDLE\_VALUE) {

cout << "Invalid read file" << endl;

return -1;

}

HANDLE handleWrite = CreateFileA("out.txt", GENERIC\_WRITE, FILE\_SHARE\_WRITE, nullptr, OPEN\_EXISTING, FILE\_FLAG\_NO\_BUFFERING | FILE\_FLAG\_OVERLAPPED , nullptr);

if (handleWrite == INVALID\_HANDLE\_VALUE) {

cout << "Invalid write file" << endl;

return -1;

}

LARGE\_INTEGER cur\_pos;

DWORD tOperCount = operCount - 1;

int sync\_times = 0;

DWORD src\_size = GetFileSize(handleRead, NULL);

const DWORD ovrCount = src\_size % SizeCluster == 0 ? src\_size / SizeCluster : src\_size / SizeCluster + 1;

vector<OVERLAPPED> ovrVector(ovrCount);

for (DWORD i = 0; i <ovrCount; i++) {

ovrVector.at(i).hEvent = handleWrite;

}

cout << "Src size: " << src\_size << " Overlapped\_count: " << ovrCount << endl;

info = new char\* [ovrCount];

for (size\_t i = 0; i < ovrCount; i++) {

info[i] = new char[SizeCluster];

}

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

for (size\_t i = 0; i < ovrCount; i++) {

cur\_pos.QuadPart = i \* SizeCluster;

ovrVector.at(i).Offset = cur\_pos.LowPart;

ovrVector.at(i).OffsetHigh = cur\_pos.HighPart;

DWORD t = ReadFileEx(handleRead, info[i], SizeCluster, &ovrVector.at(i), (LPOVERLAPPED\_COMPLETION\_ROUTINE)CopyFinal);

if (i == tOperCount) {

sync\_times++;

tOperCount += operCount;

SleepEx(INFINITE, TRUE);

}

}

if (sync\_times \* operCount < ovrCount && operCount != 0) {

sync\_times++;

SleepEx(INFINITE, TRUE);

}

while (ovrCount != write\_time) SleepEx(100, TRUE);

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

auto time = chrono::duration\_cast<std::chrono::duration<double>>(end - start).count();

LARGE\_INTEGER src\_size\_quad;

src\_size\_quad.QuadPart = src\_size;

BOOL t = SetFilePointer(handleWrite, src\_size\_quad.LowPart, &src\_size\_quad.HighPart, 0);

if (!t) {

cout << "SetFilePointer ERROR" << endl;

}

t = SetEndOfFile(handleWrite);

if (!t) {

cout << "SetEndOfFile ERROR" << endl;

}

cout << "read file size (kb): " << GetFileSize(handleRead, NULL) / 1024 << endl;

cout << "write file size (kb): " << GetFileSize(handleWrite, NULL) / 1024 << endl;

cout << "Time (in seconds): " << time << endl;

CloseHandle(handleRead);

CloseHandle(handleWrite);

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

delete [] info[i];

delete []info;

}

int main()

{

cout << "Press multiplier claster: ";

cin >> mul;

SizeCluster = 4096 \* mul;

//cout << "\nPress Number operations: ";

//cin >> operCount;

operCount = 8;

cout << endl;

BlockCopying();

ifstream fin1("testfile.txt"), fin2("out.txt");

bool f = true;

char x, y;

int i = 0;

while (f && !fin1.eof() && !fin2.eof()) {

fin1 >> x;

fin2 >> y;

if (x == y)

i++;

else

f = false;

}

if (f)

if (fin1.eof() && fin2.eof())

cout << "File equalante";

else

cout << "Diffrent size";

else

cout << i << " " << x << " " << y;

system("pause");

}