**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**ЛАБОРАТОРНАЯ РАБОТА №1**

**по дисциплине «Операционные системы»**

Тема**: “Управление файловой системой”**

| Студент |  | . |
| --- | --- | --- |
| Преподаватель |  | Тимофеев А. В. |

Санкт-Петербург

2023

**Цель работы:** исследовать управление файловой системой с помощью Win32 API.

**Задание 1.1.** Управление дисками, каталогами и файлами.

1. Вывод списка дисков:

Функция GetLogicalDrives в Windows API возвращает битовую маску, представляющую логические диски, доступные в системе. Каждый бит в маске соответствует одному логическому диску. Если бит установлен в 1, это означает, что диск определен и доступен, а если бит равен 0, диск отсутствует или недоступен.

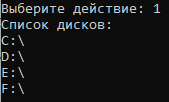


Рисунок 1

1. для одного из выбранных дисков вывод информации о диске и размер свободного пространства:

Функция GetDriveType возвращает тип указанного логического диска. Тип диска определяется на основе типа носителя, на котором находится диск, например, жесткий диск, CD-ROM, съемный диск и т. д.

Функция GetVolumeInformation возвращает информацию о файловой системе, объеме, и именах томов указанного диска.

Функция GetDiskFreeSpace возвращает информацию о свободном пространстве на диске, указанном по корневому пути

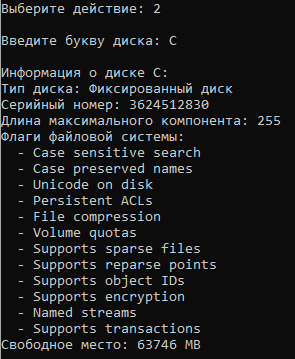


Рисунок 2

1. создание и удаление заданных каталогов:

Функция CreateDirectory используется для создания нового каталога по указанному пути. Она позволяет программе создавать новые директории, которые могут быть использованы для хранения файлов, подкаталогов и других данных.

RemoveDirectory — это функция, которая используется для удаления каталога.

3.1) Создание нового каталога:



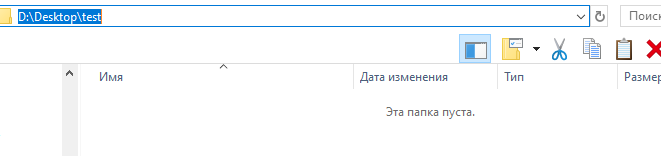


Рисунок 3

3.2) Создание каталога, который уже существует:



Рисунок 4

3.3) Удаление каталога:



Рисунок 5

3.4) Удаление каталога, которого не существует:



Рисунок 6

1. создание файлов в новых каталогах:

Функция CreateFile используется для создания или открытия файла.

4.1) Создание файла:



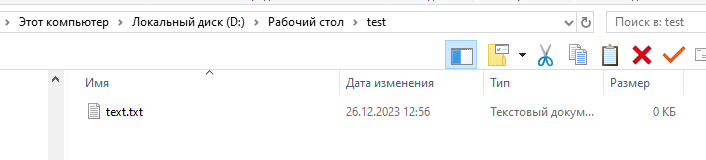


Рисунок 7

4.2) Создание файла без указания расширения:



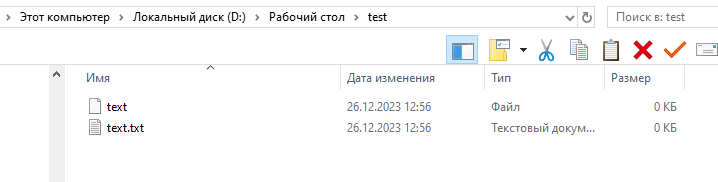


Рисунок 8

4.3) Создание файла, который уже существует:



Рисунок 9

1. копирование и перемещение файлов между каталогами с возможностью выявления попытки работы с файлами, имеющими совпадающие имена:

Функция MoveFile используется для перемещения файла или каталога в операционной системе Windows. Она позволяет программе изменить местоположение файла или каталога.

Функция CopyFile используется для копирования файла в операционной системе Windows. Она позволяет программе создать копию исходного файла в указанном местоположении.

5.1) Копирование файла:



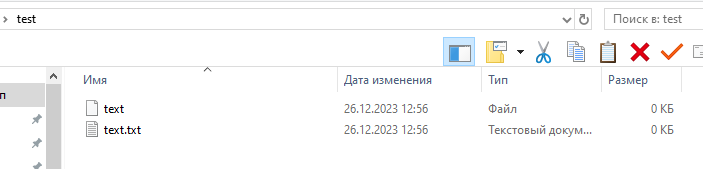


Рисунок 10

5.2) Копирование файла, который уже существует в папке назначения:



Рисунок 11

5.3) Перемещение файла:



Рисунок 12

5.4) Перемещение файла, который уже существует в папке назначения:

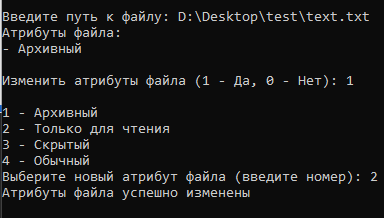


Рисунок 13

1. анализ и изменение атрибутов файлов:

Функция GetFileAttributes используется для получения атрибутов файла в операционной системе Windows. Она позволяет программе получить информацию о различных атрибутах файла, таких как атрибуты доступа, тип файла, дата и время создания, модификации и доступа, и другие.

6.1) изменение атрибутов:



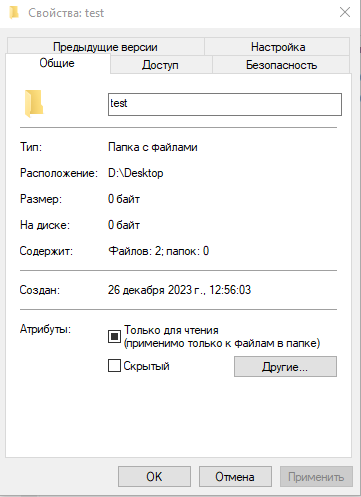
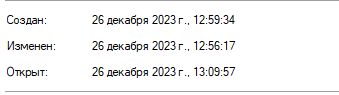
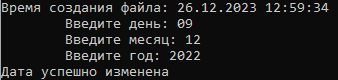
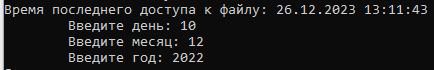


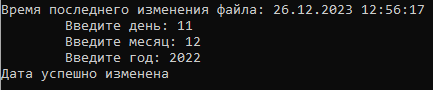
Рисунок 14

6.2) изменение времени:









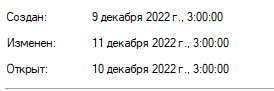


Рисунок 15

**Вывод:**

Для разработки приложений под операционную систему Windows изучение управления файловой системой с помощью Win32 API является важным аспектом. Win32 API предоставляет мощные функции и возможности для работы с файлами, папками и другими объектами файловой системы.

Использование Win32 API позволяет получить полный контроль над операциями с файлами и папками. Благодаря широкому спектру функций, доступных в Win32 API, можно создавать, копировать, перемещать и удалять файлы, а также читать, записывать и изменять атрибуты файлов. Это дает разработчикам возможность создавать гибкие и функциональные приложения, отвечающие требованиям.

Преимущество использования Win32 API состоит в его высокой производительности при работе с файлами и папками. Он предоставляет оптимизированные функции, позволяющие выполнять операции с высокой скоростью доступа к данным. Такая эффективность особенно важна при работе с большими объемами данных или приложениями, требующими высокой производительности.

Еще одним преимуществом Win32 API является возможность обработки ошибок и исключений, связанных с операциями файловой системы. Разработчики могут использовать функции Win32 API для проверки наличия файлов и папок, обработки недопустимых операций и управления исключительными ситуациями, такими как отсутствие прав доступа или нехватка места на диске.

Win32 API обеспечивает гибкость, высокую производительность, надежность и интеграцию с другими компонентами операционной системы Windows. Благодаря этому разработчики могут создавать высококачественные приложения, эффективно управляющие файловой системой и отвечающие потребностям пользователей.

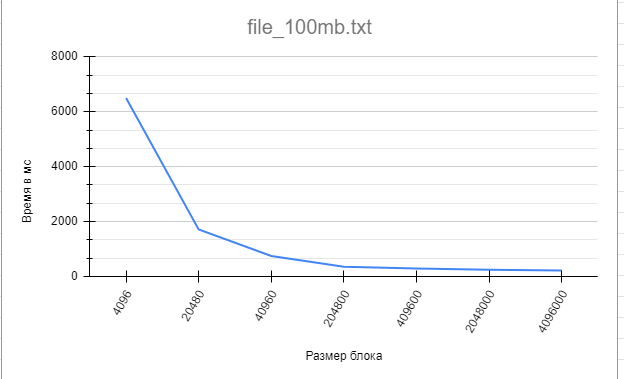
**Задание 1.2.** Копирование файла с помощью операций перекрывающегося ввода-вывода.

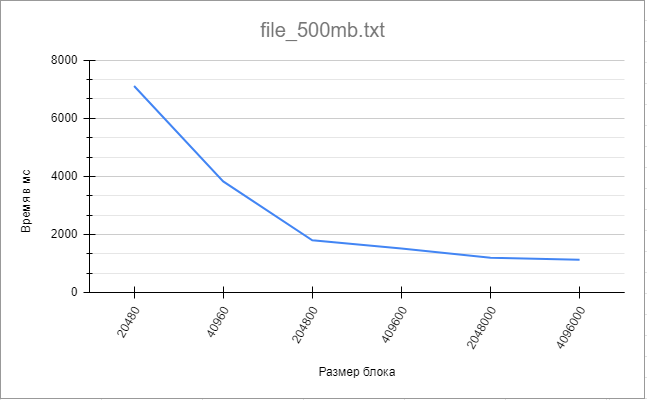
Приложение должно копировать существующий файл в новый файл, «одновременно» выполняя n перекрывающихся операций ввода-вывода блоками данных кратными размеру кластера.

1. Подберем оптимальный размер блока для разного размера копируемых файлов для ситуации с перекрывающимся выполнением одной операции ввода и одной операции вывода

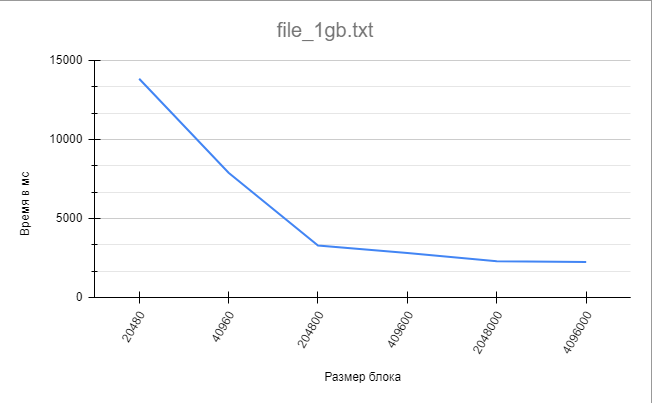
Размеры файлов: 100 мб, 500 мб, 1 гб, 2 гб, 5 гб.

Графики зависимостей времени копирования от размера блока:

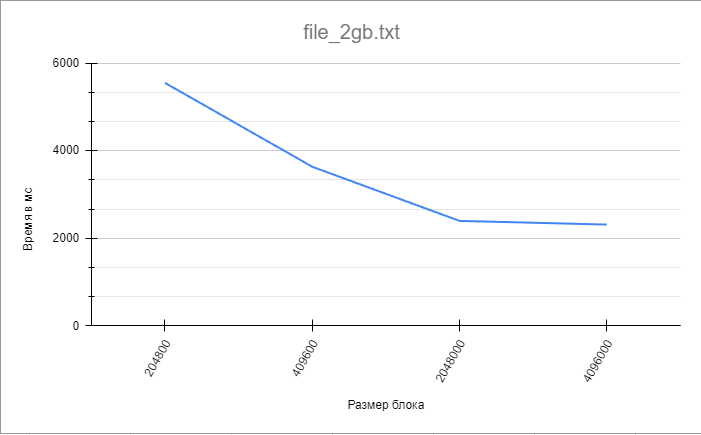




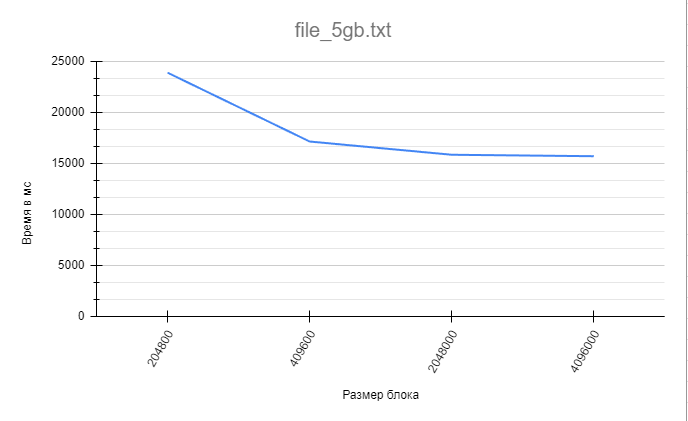
При попытке поставить размер блока равным 4096, выдается ошибка 133.



При попытке поставить размер блока равным 4096, выдается ошибка 133.



При попытке поставить размер блока равным 4096 и 20480, выдается ошибка 133.

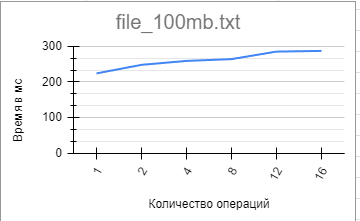


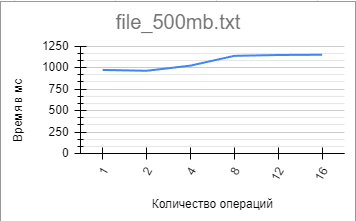
При попытке поставить размер блока равным 4096, 20480 и 40960, выдается ошибка 133.

Копирование большими блоками многократно сокращает время выполнения операций копирования для файлов любого размера.

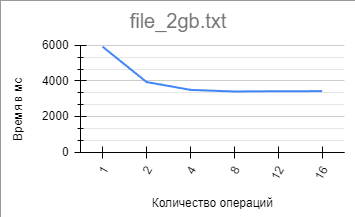
В результате проведённого эксперимента можно сделать вывод, что оптимальным размером блока является 409600 байт (т.к. при большем размере блока программа выдает ошибку при попытке работы с несколькими перекрывающимися операциями), так как при таком размере достигается наибольшая скорость для файлов всех размеров.

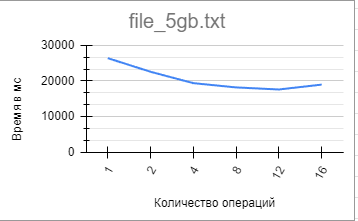
Теперь установим оптимальный размер блока в 409600 байт и будем изменять количество перекрывающих операций:











| **Размер** | **100мб** | **500мб** | **1гб** | **2гб** | **5гб** |
| --- | --- | --- | --- | --- | --- |
| **1** | 224 | 974 | 3059 | 5919 | 26397 |
| **2** | 248 | 964 | 2383 | 3943 | 22586 |
| **4** | 259 | 1024 | 2109 | 3504 | 19403 |
| **8** | 264 | 1138 | 2058 | 3413 | 18204 |
| **12** | 285 | 1148 | 2067 | 3427 | 17626 |
| **16** | 287 | 1151 | 2073 | 3431 | 18987 |

Графики показывают, что с увеличением размера копируемого файла эффективность перекрывающих операций возрастает. Для файлов размером от 500 МБ до 2 ГБ оптимальным количеством перекрывающих операций ввода-вывода является 8, в то время как для файлов меньшего размера (до 500 МБ) оптимально использовать 1-2 операции, а для файлов большего размера (5 ГБ) - 12 операций.

При работе с большими файлами процесс перекрывающих операций ввода-вывода может быть более эффективным. В больших файлах часто используется разделение на блоки данных, чтобы обеспечить более эффективную обработку. Количество блоков данных, которые можно обработать, зависит от доступного количества перекрывающих операций ввода-вывода. Таким образом, при использовании 8 операций ввода-вывода на файлах размером от 500 МБ мы можем обрабатывать 8 блоков данных, что значительно ускоряет процесс обработки.

Когда мы имеем дело с файлами меньшего размера (до 500 МБ), ограничения на количество одновременно обрабатываемых блоков данных становятся значительными. Причина в том, что для каждой операции ввода-вывода требуется дополнительное время на инициализацию и завершение операции. В связи с этим, для достижения оптимальной производительности может быть достаточно всего 1-2 операций ввода-вывода. Если использовать большее количество операций, это может добавить лишнюю нагрузку на систему из-за накладных расходов на управление этими операциями и даже ухудшить производительность.

Итак, чтобы достичь наилучшей эффективности перекрывающих операций ввода-вывода, важно учитывать объем файла и выбирать оптимальное количество операций для данного размера файла.

Ограничение количества одновременно выполняемых операций при копировании файлов размером более 500 МБ может быть обусловлено несколькими причинами. Во-первых, некоторые устройства хранения данных могут иметь ограничения на скорость чтения и записи, и достижение оптимальной пропускной способности может потребовать ограничения числа одновременно выполняемых операций. Это связано с тем, что при увеличении количества операций может произойти насыщение пропускной способности устройства, и дальнейшее увеличение числа операций не приведет к улучшению скорости копирования.

Во-вторых, операционные системы и файловые системы могут проводить оптимизации для эффективного использования системных ресурсов. Это может включать ограничение числа одновременных операций ввода-вывода для предотвращения конкуренции за ресурсы и достижения лучшей производительности. Такие ограничения могут быть настроены на оптимальное количество операций, которые максимально используют доступные ресурсы и не вызывают лишней нагрузки на систему.

Итак, ограничение числа перекрывающих операций при копировании файлов большого размера может быть связано как с физическими ограничениями устройств хранения данных, так и с оптимизациями операционной системы и файловой системы для достижения наилучшей производительности.

Перекрывающие операции ввода-вывода или асинхронные операции ввода-вывода - это метод, который позволяет выполнить операции ввода-вывода параллельно с другими задачами в программе. Вместо того чтобы ожидать завершения операции, и блокировать выполнение программы, программа может запросить операцию и продолжить свое выполнение, а система уведомит программу о завершении операции, когда она будет выполнена. Это позволяет программам быть более эффективными, использовать ресурсы процессора эффективнее и повысить общую производительность.

Традиционные операции ввода-вывода, такие как чтение данных из файла или запись на диск, обычно являются блокирующими. Когда программа выполняет такую операцию, она приостанавливается и ожидает завершения операции, прежде чем продолжить свою работу. В то время как программа ожидает, она не может выполнять другие задачи, что может быть нежелательным в случае, когда есть много других задач, которые нужно выполнить.

С помощью перекрывающих операций ввода-вывода, программа может запросить начало операции и продолжить свое выполнение. Операционная система будет следить за состоянием операции и уведомит программу, когда она будет завершена. Это позволяет программе мгновенно переключаться между различными задачами, не блокируя ее выполнение на длительное время в ожидании результата операции ввода-вывода. Такой подход особенно полезен при работе с большими файлами или операциями, которые могут занимать значительное время, таким образом, программа может продолжать выполнять другие задачи, пока ожидает завершения операции ввода-вывода.

Флаги файловой системы определяют набор функций и свойств, которые поддерживаются системой для работы с файлами и каталогами. Каждый флаг имеет свою специфическую функцию:

1. Регистрозависимый поиск: Когда этот флаг включен, файловая система различает буквы верхнего и нижнего регистра при поиске файлов и каталогов. Это означает, что файлы с разным регистром будут считаться различными.

2. Сохранение регистра имен: Этот флаг указывает, что файловая система сохраняет регистр имен файлов и каталогов. Если файл был сохранен с определенным регистром, то его имя будет отображаться с сохраненным регистром.

3. Поддержка Юникода на диске: Когда этот флаг включен, файловая система поддерживает кодировку Юникода для имен файлов и содержимого файлов. Это позволяет использовать различные языки и символы в именах файлов и их содержимом.

4. Постоянные списки контроля доступа (ACL): Если этот флаг включен, файловая система поддерживает установку постоянных списков контроля доступа (ACL) для файлов и каталогов. ACL позволяет определять различные права доступа для разных пользователей или групп пользователей.

5. Сжатие файлов: Этот флаг указывает, что файловая система поддерживает сжатие файлов. Сжатие позволяет уменьшить размер файлов на диске, что может быть полезно для экономии места. Флаг компрессии файла в операционной системе означает, что файл будет сжат для экономии места на диске или для увеличения скорости передачи данных. Когда флаг компрессии включен для файла, операционная система использует алгоритм сжатия для уменьшения его размера. При записи данных в сжатый файл, операционная система применяет алгоритм компрессии к данным, чтобы уменьшить их размер перед сохранением на диске. Когда файл требуется прочитать, операционная система автоматически распаковывает данные до их исходного состояния. Флаг компрессии файла может быть полезным в случаях, когда нужно сохранить большой объем данных, но имеется ограниченное пространство на диске. Кроме того, сжатие файла может сэкономить пропускную способность при передаче данных по сети.

6. Объемные квоты (квоты на использование дискового пространства): Этот флаг указывает, что файловая система позволяет устанавливать ограничения на использование дискового пространства для конкретных пользователей или групп пользователей. Квоты позволяют ограничивать объем данных, которыми могут пользоваться пользователи или группы.

7. Поддержка разреженных файлов: Этот флаг указывает на то, что файловая система поддерживает использование разреженных файлов. Разреженные файлы хранят только ненулевые данные, что позволяет существенно экономить дисковое пространство при работе с большими файлами, содержащими множество пустых областей.

8. Точки возврата: Этот флаг указывает на поддержку файловой системой использования точек возврата. Точки возврата - это специальные метаданные, которые позволяют управлять поведением системы при обращении к определенным файлам или каталогам.

9. Идентификаторы объектов: Этот флаг указывает на то, что файловая система поддерживает использование уникальных идентификаторов для файлов и каталогов. Это может быть полезно для идентификации файлов и каталогов в случае перемещения или переименования.

10. Шифрование: Этот флаг указывает на поддержку файловой системой шифрования файлов и каталогов. Шифрование позволяет защищать данные от несанкционированного доступа путем их зашифрования.

11. Именованные потоки: Этот флаг указывает на поддержку файлаовой системой создания и использования именованных потоков. Именованные потоки позволяют хранить дополнительные данные или атрибуты файлов, не затрагивая основное содержимое файлов.

12. Транзакции: Этот флаг указывает на поддежку файловой системой использования транзакций при выполнении операций с файлами и каталогами. Транзакции обеспечивают целостность операций и позволяют отменять изменения в случае ошибок или сбоев.

**Код программ:**

**Windows:**

#include <stdio.h>  
#include <windows.h>  
#include <locale.h>  
#include <shlwapi.h>  
#include <sys/stat.h>  
#include <time.h>  
  
void printDriveList() {  
 DWORD drivesMask = GetLogicalDrives();  
 printf("Список дисков:\n");  
  
 for (int i = 0; i < 26; i++) {  
 if ((drivesMask >> i) & 1) {  
 char driveLetter = 'A' + i;  
 printf("%c:\\\n", driveLetter);  
 }  
 }  
}  
  
void getDriveType(char\* drive) {  
 UINT type = GetDriveType(drive);  
 printf("Тип диска: ");  
 switch (type) {  
 case DRIVE\_UNKNOWN:  
 printf("Неизвестный\n");  
 break;  
 case DRIVE\_NO\_ROOT\_DIR:  
 printf("Нет корневого каталога\n");  
 break;  
 case DRIVE\_REMOVABLE:  
 printf("Съемный диск\n");  
 break;  
 case DRIVE\_FIXED:  
 printf("Фиксированный диск\n");  
 break;  
 case DRIVE\_REMOTE:  
 printf("Удаленный диск\n");  
 break;  
 case DRIVE\_CDROM:  
 printf("CD/DVD диск\n");  
 break;  
 case DRIVE\_RAMDISK:  
 printf("RAM диск\n");  
 break;  
 default:  
 printf("Неизвестный тип диска\n");  
 break;  
 }  
}  
  
void printFileSysFlags(DWORD dwFileSysFlags) {  
 struct FlagDefine {  
 DWORD flag;  
 const char\* description;  
 };  
  
 static const struct FlagDefine file\_sys\_flags[] = {  
 { FILE\_CASE\_SENSITIVE\_SEARCH, "Case sensitive search" },  
 { FILE\_CASE\_PRESERVED\_NAMES, "Case preserved names" },  
 { FILE\_UNICODE\_ON\_DISK, "Unicode on disk" },  
 { FILE\_PERSISTENT\_ACLS, "Persistent ACLs" },  
 { FILE\_FILE\_COMPRESSION, "File compression" },  
 { FILE\_VOLUME\_QUOTAS, "Volume quotas" },  
 { FILE\_SUPPORTS\_SPARSE\_FILES, "Supports sparse files" },  
 { FILE\_SUPPORTS\_REPARSE\_POINTS, "Supports reparse points" },  
 { FILE\_SUPPORTS\_REMOTE\_STORAGE, "Supports remote storage" },  
 { FILE\_VOLUME\_IS\_COMPRESSED, "Volume is compressed" },  
 { FILE\_SUPPORTS\_OBJECT\_IDS, "Supports object IDs" },  
 { FILE\_SUPPORTS\_ENCRYPTION, "Supports encryption" },  
 { FILE\_NAMED\_STREAMS, "Named streams" },  
 { FILE\_READ\_ONLY\_VOLUME, "Read-only volume" },  
 { FILE\_SEQUENTIAL\_WRITE\_ONCE, "Sequential write once" },  
 { FILE\_SUPPORTS\_TRANSACTIONS, "Supports transactions" },  
 };  
  
 printf("Флаги файловой системы:\n");  
  
 for (int i = 0; i < sizeof(file\_sys\_flags) / sizeof(struct FlagDefine); i++) {  
 if (dwFileSysFlags & file\_sys\_flags[i].flag) {  
 printf(" - %s\n", file\_sys\_flags[i].description);  
 }  
 }  
}  
  
void getDriveCommonInfo(char\* drive) {  
 char szVolName[MAX\_PATH + 1], szFileSysName[MAX\_PATH + 1];  
 DWORD dwSerialNumber, dwMaxComponentLen, dwFileSysFlags;  
 GetVolumeInformation(drive, szVolName, MAX\_PATH,  
 &dwSerialNumber, &dwMaxComponentLen,  
 &dwFileSysFlags, szFileSysName, sizeof(szFileSysName));  
 printf("Серийный номер: %lu\n", dwSerialNumber);  
 printf("Длина максимального компонента: %lu\n", dwMaxComponentLen);  
 printFileSysFlags(dwFileSysFlags);  
}  
  
void getDriveFreeSpace(char\* drive) {  
 DWORD sectorsPerCluster;  
 DWORD bytesPerSector;  
 DWORD freeClusters;  
 DWORD totalClusters;  
  
 GetDiskFreeSpace(  
 drive,  
 &sectorsPerCluster,  
 &bytesPerSector,  
 &freeClusters,  
 &totalClusters  
 );  
  
 unsigned int kBPerCluster = bytesPerSector \* sectorsPerCluster / 1024;  
  
 printf("Свободное место: %lu MB\n", kBPerCluster \* freeClusters / 1024);  
}  
  
void printDiskInfo(char driveLetter) {  
 char rootPath[4];  
 sprintf(rootPath, "%c:\\", driveLetter);  
  
 printf("\nИнформация о диске %c:\n", driveLetter);  
  
 getDriveType(rootPath);  
 getDriveCommonInfo(rootPath);  
 getDriveFreeSpace(rootPath);  
}  
  
void createDirectory() {  
 printf("\nВведите путь к каталогу, который хотите создать: ");  
 char directoryPath[MAX\_PATH];  
 scanf("%s", directoryPath);  
 if (CreateDirectory(directoryPath, NULL)) {  
 printf("Каталог успешно создан\n");  
 } else {  
 DWORD errorCode = GetLastError();  
 if (errorCode == ERROR\_ALREADY\_EXISTS) {  
 printf("Каталог уже существует\n");  
 } else {  
 printf("Произошла ошибка при создании каталога. Код ошибки: %lu\n", errorCode);  
 }  
 }  
}  
  
void removeDirectory() {  
 printf("\nВведите путь к каталогу, который хотите удалить: ");  
 char directoryPath[MAX\_PATH];  
 scanf("%s", directoryPath);  
  
 if (RemoveDirectory(directoryPath)) {  
 printf("Каталог успешно удален\n");  
 } else {  
 DWORD errorCode = GetLastError();  
 if (errorCode == ERROR\_DIR\_NOT\_EMPTY) {  
 printf("Каталог не пустой. Удаление невозможно\n");  
 } else if (errorCode == ERROR\_PATH\_NOT\_FOUND) {  
 printf("Указанный каталог не найден\n");  
 } else {  
 printf("Произошла ошибка при удалении каталога. Код ошибки: %lu\n", errorCode);  
 }  
 }  
}  
  
void createFile() {  
 printf("\nВведите путь к файлу, который хотите создать: ");  
 char filePath[MAX\_PATH];  
 scanf("%s", filePath);  
 HANDLE fileHandle = CreateFile(filePath, GENERIC\_WRITE, 0, NULL, CREATE\_NEW, FILE\_ATTRIBUTE\_NORMAL, NULL);  
 if (fileHandle != INVALID\_HANDLE\_VALUE) {  
 printf("Файл успешно создан\n");  
 CloseHandle(fileHandle);  
 } else {  
 DWORD errorCode = GetLastError();  
 if (errorCode == ERROR\_FILE\_EXISTS) {  
 printf("Файл уже существует\n");  
 } else {  
 printf("Произошла ошибка при создании файла. Код ошибки: %lu\n", errorCode);  
 }  
 }  
}  
  
void copyMoveFile(BOOL isMoveOperation) {  
 printf("\nВведите путь к исходному файлу: ");  
 char sourcePath[MAX\_PATH];  
 scanf("%s", sourcePath);  
  
 printf("Введите путь к целевому каталогу: ");  
 char targetDirectory[MAX\_PATH];  
 scanf("%s", targetDirectory);  
  
  
 char targetPath[MAX\_PATH];  
 snprintf(targetPath, sizeof(targetPath), "%s\\%s", targetDirectory, PathFindFileName(sourcePath));  
  
 if (isMoveOperation) {  
 if (MoveFile(sourcePath, targetPath)) {  
 printf("Файл успешно перемещен\n");  
 } else {  
 DWORD errorCode = GetLastError();  
 printf("Произошла ошибка при перемещении файла. Код ошибки: %lu\n", errorCode);  
 }  
 } else {  
 if (CopyFile(sourcePath, targetPath, FALSE)) {  
 printf("Файл успешно скопирован\n");  
 } else {  
 DWORD errorCode = GetLastError();  
 printf("Произошла ошибка при копировании файла. Код ошибки: %lu\n", errorCode);  
 }  
 }  
}  
  
void analyzeAndModifyFileAttributes() {  
 printf("\nВведите путь к файлу: ");  
 char filePath[MAX\_PATH];  
 scanf("%s", filePath);  
  
 DWORD fileAttributes = GetFileAttributes(filePath);  
 if (fileAttributes == INVALID\_FILE\_ATTRIBUTES) {  
 DWORD errorCode = GetLastError();  
 printf("Произошла ошибка при получении атрибутов файла. Код ошибки: %lu\n", errorCode);  
 return;  
 }  
  
 printf("Атрибуты файла:\n");  
  
 if (fileAttributes & FILE\_ATTRIBUTE\_READONLY) {  
 printf("- Только для чтения\n");  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_HIDDEN) {  
 printf("- Скрытый\n");  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_DIRECTORY) {  
 printf("- Каталог\n");  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_ARCHIVE) {  
 printf("- Архивный\n");  
 }  
 if (fileAttributes & FILE\_ATTRIBUTE\_NORMAL) {  
 printf("- Обычный\n");  
 }  
  
  
 printf("\nИзменить атрибуты файла (1 - Да, 0 - Нет): ");  
 int modifyAttributes;  
 scanf("%d", &modifyAttributes);  
  
 if (modifyAttributes) {  
 DWORD newFileAttributes = fileAttributes;  
  
 printf("\n1 - Архивный\n"  
 "2 - Только для чтения\n"  
 "3 - Скрытый\n"  
 "4 - Обычный\n"  
  
 "Выберите новый атрибут файла (введите номер): ");  
 int attributeSelection;  
 scanf("%d", &attributeSelection);  
  
 switch (attributeSelection) {  
 case 1:  
 if (fileAttributes & FILE\_ATTRIBUTE\_ARCHIVE) {  
 newFileAttributes &= ~FILE\_ATTRIBUTE\_ARCHIVE;  
 } else {  
 newFileAttributes |= FILE\_ATTRIBUTE\_ARCHIVE;  
 }  
 break;  
 case 2:  
 if (fileAttributes & FILE\_ATTRIBUTE\_READONLY)  
 newFileAttributes &= ~FILE\_ATTRIBUTE\_READONLY;  
 else  
 newFileAttributes |= FILE\_ATTRIBUTE\_READONLY;  
 break;  
 case 3:  
 if (fileAttributes & FILE\_ATTRIBUTE\_HIDDEN)  
 newFileAttributes &= ~FILE\_ATTRIBUTE\_HIDDEN;  
 else  
 newFileAttributes |= FILE\_ATTRIBUTE\_HIDDEN;  
 break;  
 case 4:  
 newFileAttributes = FILE\_ATTRIBUTE\_NORMAL;  
 break;  
 default:  
 printf("Неправильный выбор. Атрибуты не будут изменены\n");  
 return;  
 }  
  
 if (SetFileAttributes(filePath, newFileAttributes)) {  
 printf("Атрибуты файла успешно изменены\n");  
 } else {  
 DWORD errorCode = GetLastError();  
 printf("Произошла ошибка при изменении атрибутов файла. Код ошибки: %lu\n", errorCode);  
 }  
 }  
}  
  
void printFileCreationTime(const char\* path) {  
 struct stat attrib;  
 if (stat(path, &attrib) == 0) {  
 struct tm\* creationTime = (struct tm \*) localtime(&attrib.st\_ctime);  
 char timeString[100];  
 strftime(timeString, sizeof(timeString), "%c", creationTime);  
 printf("Время создания файла: %s\n", timeString);  
 } else {  
 printf("Ошибка при получении информации о файле\n");  
 }  
}  
  
void printFileLastAccessTime(const char\* path) {  
 struct stat attrib;  
 if (stat(path, &attrib) == 0) {  
 struct tm\* LastAccessTime = (struct tm \*) localtime(&attrib.st\_atime);  
 char timeString[100];  
 strftime(timeString, sizeof(timeString), "%c", LastAccessTime);  
 printf("Время последнего доступа к файлу: %s\n", timeString);  
 } else {  
 printf("Ошибка при получении информации о файле\n");  
 }  
}  
  
void printFileLastModifyTime(const char\* path) {  
 struct stat attrib;  
 if (stat(path, &attrib) == 0) {  
 struct tm\* LastModifyTime = (struct tm \*) localtime(&attrib.st\_mtime);  
 char timeString[100];  
 strftime(timeString, sizeof(timeString), "%c", LastModifyTime);  
 printf("Время последнего изменения файла: %s\n", timeString);  
 } else {  
 printf("Ошибка при получении информации о файле\n");  
 }  
}  
  
void setTime() {  
  
 printf("\nВведите путь к файлу: ");  
 char path[MAX\_PATH];  
 scanf("%s", path);  
  
 if (access(path, F\_OK) != 0) {  
 printf("Ошибка: неправильный путь к файлу\n");  
 return;  
 }  
  
 int choiceTime = 0;  
 printf("Выберите, какое время хотите изменить.\n1 - создания\n2 - последнего доступа\n3 - последнего изменения\n");  
 scanf("%d", &choiceTime);  
  
 if (choiceTime == 1) {  
 SYSTEMTIME sysCreationTime = { 0 };  
 printf("Время создания\n");  
 printFileCreationTime(path);  
  
 printf("\tВведите день: ");  
 scanf("%hu", &sysCreationTime.wDay);  
 printf("\tВведите месяц: ");  
 scanf("%hu", &sysCreationTime.wMonth);  
 printf("\tВведите год: ");  
 scanf("%hu", &sysCreationTime.wYear);  
  
 FILETIME fileCreationTime;  
 SystemTimeToFileTime(&sysCreationTime, &fileCreationTime);  
  
 HANDLE handlerFile = CreateFileA(path, GENERIC\_WRITE, FILE\_SHARE\_WRITE, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);  
 if (handlerFile != INVALID\_HANDLE\_VALUE) {  
 if (SetFileTime(handlerFile, &fileCreationTime, NULL, NULL)) {  
 printf("Дата успешно изменена\n");  
 printFileCreationTime(path);  
 } else {  
 DWORD err = GetLastError();  
 printf("Ошибка при изменении информации времени\n%lu\n", err);  
 }  
 CloseHandle(handlerFile);  
 }  
 choiceTime = 0;  
 }  
  
 if (choiceTime == 2) {  
 SYSTEMTIME sysTime = { 0 };  
 printf("Дата последнего доступа\n");  
 printFileLastAccessTime(path);  
 printf("\tВведите день: ");  
 scanf("%hu", &sysTime.wDay);  
 printf("\tВведите месяц: ");  
 scanf("%hu", &sysTime.wMonth);  
 printf("\tВведите год: ");  
 scanf("%hu", &sysTime.wYear);  
 FILETIME fileTime;  
 SystemTimeToFileTime(&sysTime, &fileTime);  
 HANDLE handlerFile = CreateFileA(path, GENERIC\_WRITE, FILE\_SHARE\_WRITE, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);  
 if (handlerFile != INVALID\_HANDLE\_VALUE) {  
 if (SetFileTime(handlerFile, NULL, &fileTime, NULL)) {  
 printf("Дата успешно изменена\n");  
 } else {  
 DWORD err = GetLastError();  
 printf("Ошибка при изменении информации времени\n%lu\n", err);  
 }  
 CloseHandle(handlerFile);  
 }  
 choiceTime = 0;  
 }  
  
 if (choiceTime == 3) {  
 SYSTEMTIME sysTime = { 0 };  
 printf("Дата последнего изменения\n");  
  
 printFileLastModifyTime(path);  
  
 printf("\tВведите день: ");  
 scanf("%hu", &sysTime.wDay);  
  
 printf("\tВведите месяц: ");  
 scanf("%hu", &sysTime.wMonth);  
  
 printf("\tВведите год: ");  
 scanf("%hu", &sysTime.wYear);  
  
 FILETIME fileTime;  
 SystemTimeToFileTime(&sysTime, &fileTime);  
  
 HANDLE handlerFile = CreateFileA(path, GENERIC\_WRITE, FILE\_SHARE\_WRITE, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);  
 if (handlerFile != INVALID\_HANDLE\_VALUE) {  
 if (SetFileTime(handlerFile, NULL, NULL, &fileTime)) {  
 printf("Дата успешно изменена\n");  
 } else {  
 DWORD err = GetLastError();  
 printf("Ошибка при изменении информации времени\n%lu\n", err);  
 }  
 CloseHandle(handlerFile);  
 }  
  
 choiceTime = 0;  
 }  
}  
  
int main() {  
 setlocale(LC\_ALL, "Rus");  
 int choice;  
  
 do {  
 printf("\nМеню:\n");  
 printf("1. Вывод списка дисков\n");  
 printf("2. Вывод информации о диске и размере свободного пространства\n");  
 printf("3. Создание каталога\n");  
 printf("4. Удаление каталога\n");  
 printf("5. Создание файла\n");  
 printf("6. Копирование файла\n");  
 printf("7. Перемещение файла\n");  
 printf("8. Анализ и изменение атрибутов файла\n");  
 printf("9. Изменение времени файла\n");  
 printf("0. Выход\n");  
 printf("Выберите действие: ");  
 scanf("%d", &choice);  
  
 switch (choice) {  
 case 1:  
 printDriveList();  
 break;  
 case 2:  
 printf("\nВведите букву диска: ");  
 char driveLetter;  
 scanf(" %c", &driveLetter);  
 printDiskInfo(driveLetter);  
 break;  
 case 3:  
 createDirectory();  
 break;  
 case 4:  
 removeDirectory();  
 break;  
 case 5:  
 createFile();  
 break;  
 case 6:  
 copyMoveFile(FALSE);  
 break;  
 case 7:  
 copyMoveFile(TRUE);  
 break;  
 case 8:  
 analyzeAndModifyFileAttributes();  
 break;  
 case 9:  
 setTime();  
 break;  
 case 0:  
 printf("\nПрограмма завершена\n");  
 break;  
 default:  
 printf("\nНеправильный выбор. Попробуйте еще раз\n");  
 break;  
 }  
 } while (choice != 0);  
  
 return 0;  
}

**Linux:**

#include <iostream>  
#include <chrono>  
#include <thread>  
#include <stdio.h>  
#include <stdlib.h>  
#include <aio.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <signal.h>  
#include <errno.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <string.h>  
#include <stdint.h>  
#include <inttypes.h>  
#include <sys/statvfs.h>  
  
#define BUFFER\_SIZE 409600  
#define COUNT 12  
  
using namespace std;  
  
double ReadB = 0;  
double WriteB = 0;  
double lastWriteB = -1;  
  
struct aio\_operation  
{  
 struct aiocb aio;  
 //char \*buffer;  
 int write\_operation;  
 struct aio\_operation \*next\_operation;  
 int num;  
 bool end;  
};  
  
void aio\_completion\_handler(sigval\_t sigval)  
{  
 struct aio\_operation \*aio\_op = (struct aio\_operation \*)sigval.sival\_ptr;  
 struct aio\_operation \*next\_op = aio\_op->next\_operation;  
  
 (aio\_op->aio).aio\_offset += BUFFER\_SIZE \* COUNT;  
  
 if (aio\_op->write\_operation)  
 {  
 ssize\_t c = aio\_return(&aio\_op->aio);  
 WriteB = WriteB + c;  
 aio\_read(&next\_op->aio);  
 }  
 else  
 {  
 ssize\_t c = aio\_return(&aio\_op->aio);  
 ReadB = ReadB + c;  
 if (c != 0)  
 {  
 next\_op->aio.aio\_nbytes = c;  
 aio\_write(&next\_op->aio);  
 }  
 else  
 {  
 aio\_op->end = true;  
 }  
 }  
}  
  
int main()  
{  
 const char \*sourceFile = "text\_5GB.txt";  
 const char \*destFile = "test.txt";  
  
 double execTime = 0;  
 int execTimeInt = 0;  
  
 int sourceFd = open(sourceFile, O\_RDONLY | O\_NONBLOCK, 0666);  
 if (sourceFd == -1) {  
 perror("Error with source file");  
 return 1;  
 }  
  
 int destFd = open(destFile, O\_CREAT | O\_WRONLY | O\_TRUNC | O\_NONBLOCK, 0666);  
 if (destFd == -1) {  
 perror("Error with destination file");  
 return 1;  
 }  
  
 struct stat fileStat;  
 int checkSize = fstat(sourceFd, &fileStat);  
 if (checkSize == -1) {  
 perror("Error with size of source file");  
 return 1;  
 }  
  
 int val = fileStat.st\_size / COUNT;  
 if (fileStat.st\_size % COUNT != 0) val++;  
  
 char buffer[COUNT][BUFFER\_SIZE];  
 struct aio\_operation read\_aio\_op[COUNT];  
 struct aio\_operation write\_aio\_op[COUNT];  
  
 for (int i = 0; i < COUNT; i++)  
 {  
 memset(&read\_aio\_op[i], 0, sizeof(struct aio\_operation));  
 read\_aio\_op[i].aio.aio\_fildes = sourceFd;  
 read\_aio\_op[i].aio.aio\_buf = buffer[i];  
 read\_aio\_op[i].aio.aio\_nbytes = BUFFER\_SIZE;  
 read\_aio\_op[i].write\_operation = 0;  
 read\_aio\_op[i].aio.aio\_sigevent.sigev\_notify = SIGEV\_THREAD;  
 read\_aio\_op[i].aio.aio\_sigevent.sigev\_notify\_function = aio\_completion\_handler;  
 read\_aio\_op[i].aio.aio\_sigevent.sigev\_value.sival\_ptr = &read\_aio\_op[i];  
 read\_aio\_op[i].aio.aio\_offset = BUFFER\_SIZE \* i;  
 read\_aio\_op[i].num = i;  
  
 memset(&write\_aio\_op[i], 0, sizeof(struct aio\_operation));  
 write\_aio\_op[i].aio.aio\_fildes = destFd;  
 write\_aio\_op[i].aio.aio\_buf = buffer[i];  
 write\_aio\_op[i].aio.aio\_nbytes = BUFFER\_SIZE;  
 write\_aio\_op[i].aio.aio\_sigevent.sigev\_notify = SIGEV\_THREAD;  
 write\_aio\_op[i].aio.aio\_sigevent.sigev\_notify\_function = aio\_completion\_handler;  
 write\_aio\_op[i].aio.aio\_sigevent.sigev\_value.sival\_ptr = &write\_aio\_op[i];  
 write\_aio\_op[i].write\_operation = 1;  
 write\_aio\_op[i].aio.aio\_offset = BUFFER\_SIZE \* i;  
 write\_aio\_op[i].num = i;  
  
 read\_aio\_op[i].next\_operation = &write\_aio\_op[i];  
 write\_aio\_op[i].next\_operation = &read\_aio\_op[i];  
 }  
  
 auto begin = chrono::steady\_clock::now();  
  
 for (int i = 0; i < COUNT; i++)  
 {  
 aio\_read(&read\_aio\_op[i].aio);  
 }  
  
 while (WriteB != fileStat.st\_size)  
 {  
 this\_thread::sleep\_for(chrono::milliseconds(10));  
 }  
  
 cout << "Close files: ";  
 close(sourceFd);  
 close(destFd);  
 cout << "success" << endl;  
  
 auto end = chrono::steady\_clock::now();  
  
 auto ms = chrono::duration\_cast<chrono::milliseconds>(end - begin);  
  
 cout << "Source file: " << sourceFile << endl;  
 cout << "New file: " << destFile << endl;  
 cout << "New file size: " << fileStat.st\_size << " b" << endl;  
 cout << "Write size: " << WriteB << " b" << endl;  
 cout << "Time: " << ms.count() << " ms" << endl;  
 return 0;  
}