Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Лабораторна робота №5

«Файлова система (частина 2)»

з дисципліни
«Операційні системи»

Виконала:

студентка групи ІМ-11
Бащак Ярина Володимирівна

Перевірив:

ст. вик. Сімоненко А.В.

Київ 2023

Завдання.

Завдання на роботу

Додати до драйвера ФС, який був розроблений у лабораторній роботі №4, підтримку символічних посилань та дерева директорій. Додати в програму команди для роботи з символічними посиланнями та директоріями.

Програма повинна підтримувати шляхові імена у всіх командах (також тих, які були розроблені в лабораторній роботі №4). Якщо остання компонента шляхового імені є символічним посиланням, тоді команди `link` та `unlink` мають працювати з символічним посиланням, тобто на символічне посилання можуть вказувати кілька жорстких посилань. Команда `link` не повинна створювати жорсткі посилання на директорії. Команда `unlink` не повинна працювати з жорсткими посиланнями на директорії.

Додати до програми підтримку наступних команд (замість введення команд можна викликати необхідні функції безпосередньо в програмі):

- `mkdir pathname` – створити директорію та створити відповідне жорстке посилання на неї.
- `rmdir pathname` – звільнити порожню директорію на яке вказує шляхове ім’я та знищити відповідне жорстке посилання на неї (вміст директорії не повинен мати жодного жорсткого посилання, крім наперед визначених жорстких посилань з іменами **.** та **..**).
- `cd pathname` – змінити поточну робочу директорію.
- `symlink str pathname` – створити символічне посилання з вмістом `str` та створити на нього відповідне жорстке посилання. Максимальна довжина вмісту символічного посилання `str` не має перевищувати розмір одного блоку.

У цих командах `pathname` – це шляхове ім’я. У командах з лабораторної роботи №4 замість `name` можна записати `pathname`, для позначення того, що всі команди мають підтримувати роботу з шляховими іменами.

Все, що написано в лабораторній роботі №4, також справедливо для цієї лабораторної роботи.

Основні методи**,** які використовуються для роботи з **pathnames**, які були впроваджені, в тому числі у всі команди з попередньої лабораторної роботи:

- метод, для розділення шляху на **parentPath** і **itemName**. Наприклад, якщо з кореневої директорії, в якій є директорія dir1 застовувати наступні назви шляхів “dir1”, “/dir1”, “dir1/”, “/dir1/”, “./dir1”, “../dir1” , “./dir1/”, “../dir1/”, то вони означатимуть одне й те ж.

```
    private (string parentPath, string itemName) GetParentPathAndItemName(string
path)
    {
        string parentPath, itemName;

        if (path == "/")
        {
            parentPath = "/";
            itemName = "";
        }
        else
        {
            path = path.TrimEnd('/');

            var lastIndex = path.LastIndexOf('/');
            itemName = lastIndex == -1 ? path : path[(lastIndex + 1)..];
            parentPath = lastIndex == -1 ? "" : lastIndex == 0 ? "/" :
path[..lastIndex];
        }

        return (parentPath, itemName);
    }
```

- метод, для **отримання дескриптора** директорії за шляхом. Зазвичай сюди передається parentPath, знайдений в попередньому методі. Якщо якась частина з шляху є **символічним посиланням**, воно теж обробляється тут.

```
    public FileDescriptor GetDirectoryDescriptorByPath(string path)

    {
        int symlinkLoopCount = 0;
        FileDescriptor? startDescriptor;

        if (path.StartsWith("/"))
        {
            startDescriptor = Descriptors[0];
            path = path.TrimStart('/');
        }
        else
        {
            startDescriptor = CurrentDirectory;
        }
```

```
        if (startDescriptor == null)
        {
            throw new InvalidOperationException("Something wrong occured while
finding root or current directory.");
        }

        if (string.IsNullOrEmpty(path))
        {
            return startDescriptor;
        }

        string[] parts = path.Split(new char[] { '/' },
StringSplitOptions.RemoveEmptyEntries);
        FileDescriptor current = startDescriptor;

        foreach (var part in parts)
        {
            int index = current.Directory.FindDescriptorIndex(part);
            if (index == -1)
            {
                throw new FileNotFoundException($"Part '{part}' of the path was not
found.");
            }

            current = Descriptors[index]!;

            if (current.Type == FileType.Sym)
            {
                if (++symlinkLoopCount > MaxSymlinkLoops)
                {
                    throw new InvalidOperationException("Too many levels of
symbolic links.");
                }

                current = ResolveSymlink(current);
            }

            if (current.Type != FileType.Dir)
            {
                throw new InvalidOperationException("Path is not a directory.");
            }
        }

        return current;
    }

    private FileDescriptor ResolveSymlink(FileDescriptor symlinkDescriptor)
    {
        string targetPath = symlinkDescriptor.SymLinkTarget;

        return GetDescriptorByPath(targetPath);
    }
```

- метод для отримання **дескриптора за індексом**. Теж працює з символічними посиланнями, повертаючи дескриптор оригінального файлу/директорії.

```
    private FileDescriptor GetFileDescriptorByIndex(int descriptorIndex, bool
shouldBeReg = true)
    {
        FileDescriptor fileDescriptor = Descriptors[descriptorIndex]!;
        if (fileDescriptor.Type == FileType.Sym)
        {
            var targetPath = fileDescriptor.SymLinkTarget;
            var (targetParentPath, targetFilename) =
GetParentPathAndItemName(targetPath);
            var targetParrentDirectoryDescriptor =
GetDirectoryDescriptorByPath(targetParentPath);

            int targetDescriptorIndex =
targetParrentDirectoryDescriptor.Directory.FindDescriptorIndex(targetFilename);
            fileDescriptor = Descriptors[targetDescriptorIndex]!;
        }

        if (shouldBeReg)
        {
            if (fileDescriptor.Type != FileType.Reg)
            {
                throw new InvalidOperationException($"The path you provided is not
a regular file.");
            }
        }
        else
        {
            if (fileDescriptor.Type != FileType.Dir)
            {
                throw new InvalidOperationException($"The path you provided is not
a directory.");
            }
        }

        return fileDescriptor;
    }
```

Приклади виконання всіх команд.

- mkdir і правильна обробка різних варіантів задання шляхів у командах з минулої лабораторної

```
static void Main(string[] args)
{
    int maxNumberOfDescriptors = 10;
    var fileSystem = new FileSystem(maxNumberOfDescriptors);

    fileSystem.MakeDirectory("dir1");
    fileSystem.MakeDirectory("/dir2");
    fileSystem.MakeDirectory("/dir1/dir3");
    fileSystem.Create("/dir1/file1");
    fileSystem.Create("dir1/file2");
    fileSystem.Ls();
    fileSystem.Ls("/");
    fileSystem.Stat();
    fileSystem.Stat("/");
    fileSystem.Stat("dir1/");
    fileSystem.Stat("dir1");
    fileSystem.ChangeDirectory("dir1");
    fileSystem.Create("file3");
    fileSystem.Ls();
    fileSystem.Ls("dir3/");
    fileSystem.Stat();
    fileSystem.Stat("/");
    fileSystem.Stat("/dir1");
    fileSystem.Stat("/dir1/");
    fileSystem.Stat("dir3");
    fileSystem.Stat("file3");
    fileSystem.Stat("./file3");
    fileSystem.Stat("../file1");
    fileSystem.Stat("/dir1/file3");
```

```
$ dotnet run
File with name 'dir1' was created, descriptor index: 1
File with name 'dir2' was created, descriptor index: 2
File with name 'dir3' was created, descriptor index: 3
File with name 'file1' was created, descriptor index: 4
File with name 'file2' was created, descriptor index: 5
Directory Listing:
.       => dir, 0
..      => dir, 0
dir1    => dir, 1
dir2    => dir, 2
Directory Listing:
.       => dir, 0
..      => dir, 0
dir1    => dir, 1
dir2    => dir, 2
'' => type=dir nlink=5 size=4 items
'/' => type=dir nlink=5 size=4 items
'dir1/' => type=dir nlink=3 size=5 items
'dir1' => type=dir nlink=3 size=5 items
CWD was changed to 'dir1'.
File with name 'file3' was created, descriptor index: 6
Directory Listing:
.       => dir, 1
..      => dir, 0
dir3    => dir, 3
file1   => reg, 4
file2   => reg, 5
file3   => reg, 6
Directory Listing:
.       => dir, 3
..      => dir, 1
'' => type=dir nlink=3 size=6 items
'/' => type=dir nlink=5 size=4 items
'/dir1' => type=dir nlink=3 size=6 items
'/dir1/' => type=dir nlink=3 size=6 items
'dir3' => type=dir nlink=2 size=2 items
'file3' => type=reg nlink=1, size=0, nblock=0
'./file3' => type=reg nlink=1, size=0, nblock=0
'../dir1' => type=dir nlink=3 size=6 items
File '../file1' not found.
'/dir1/file3' => type=reg nlink=1, size=0, nblock=0
```

- cd

```
        fileSystem.Ls();
        fileSystem.ChangeDirectory("dir3");
        fileSystem.Ls();
        fileSystem.ChangeDirectory("..");
        fileSystem.Ls();
        fileSystem.ChangeDirectory("/");
        fileSystem.Ls();
```

```
Directory Listing:
.       => dir, 1
..      => dir, 0
dir3    => dir, 3
file1   => reg, 4
file2   => reg, 5
file3   => reg, 6
CWD was changed to 'dir3'.
Directory Listing:
.       => dir, 3
..      => dir, 1
CWD was changed to '..'.
Directory Listing:
.       => dir, 1
..      => dir, 0
dir3    => dir, 3
file1   => reg, 4
file2   => reg, 5
file3   => reg, 6
CWD was changed to '/'.
Directory Listing:
.       => dir, 0
..      => dir, 0
dir1    => dir, 1
dir2    => dir, 2
```

- rmdir

```
        fileSystem.Ls("dir1");
        fileSystem.RemoveDirectory("dir1");
        fileSystem.Unlink("dir1/file1");
        fileSystem.Unlink("dir1/file2");
        fileSystem.Unlink("dir1/file3");
        fileSystem.RemoveDirectory("dir1/dir3");
        fileSystem.Ls("dir1");
        fileSystem.RemoveDirectory("dir1");
```

```
Directory Listing:
.       => dir, 1
..      => dir, 0
dir3    => dir, 3
file1   => reg, 4
file2   => reg, 5
file3   => reg, 6
Directory 'dir1' is not empty. You can not remove it.
Successfully unlinked the hard link with pathname 'dir1/file1'.
Successfully released file descriptor index '4'.
Successfully unlinked the hard link with pathname 'dir1/file2'.
Successfully released file descriptor index '5'.
Successfully unlinked the hard link with pathname 'dir1/file3'.
Successfully released file descriptor index '6'.
Successfully released file descriptor index '3'.
Successfully removed directory with pathname 'dir1/dir3'.
Directory Listing:
.       => dir, 1
..      => dir, 0
Successfully released file descriptor index '1'.
Successfully removed directory with pathname 'dir1'.
```

- symlink зі звичайними файлами. Змінюючи файл за символічним посиланням, змінюється в першу чергу оригінальний файл. В прикладі нижче кожна команда працюватиме індентично, як зі шляхом "dir1/file1", так і "sym_file1".

```
        fileSystem.CreateSymlink("dir1/file1", "sym_file1");
        fileSystem.Ls();
        fileSystem.Stat("sym_file1");
        fileSystem.Stat("dir1/file1");

        var fd1 = fileSystem.Open("sym_file1");
        fileSystem.Truncate("dir1/file1", 128);
        byte[] byteArray = new byte[20];

        for (int i = 0; i < byteArray.Length; i++)
        {
            byteArray[i] = (byte)i;
        }

        fileSystem.Write(fd1, byteArray);
        fileSystem.Stat("sym_file1");
        fileSystem.Stat("dir1/file1");

        fileSystem.Close(fd1);
```

```
File with name 'sym_file1' was created, descriptor index: 6
Directory Listing:
.       => dir, 0
..      => dir, 0
dir1    => dir, 1
dir2    => dir, 2
sym_file1       => sym, 6 -> dir1/file1
'sym_file1' => type=sym nlink=1, size=0, nblock=0
'dir1/file1' => type=reg nlink=1, size=0, nblock=0
File 'sym_file1' was opened, File Descriptor: 0.
FileSize of 'file1' was increased to 128 bytes.
Successfully wrote 20 bytes to the file with descriptor number '0'.
'sym_file1' => type=sym nlink=1, size=128, nblock=1
'dir1/file1' => type=reg nlink=1, size=128, nblock=1
File was closed, now File Descriptor 0 is free.
```

- symlink з директоріями

```
        fileSystem.Ls();
        fileSystem.CreateSymlink("dir1/dir3", "dir2/sym_dir3");
        fileSystem.Ls("dir2");

        var fd1 = fileSystem.Open("dir2/sym_dir3/./file1"); // doesn't exist
        var fd2 = fileSystem.Open("dir2/sym_dir3/../file1"); // exists

        // more complex example
        fileSystem.CreateSymlink("/dir2", "dir1/sym_dir2");
        fileSystem.Stat("dir2/sym_dir3/../sym_dir2");
        fileSystem.Stat("dir2");
```

```
Directory Listing:
.       => dir, 0
..      => dir, 0
dir1    => dir, 1
dir2    => dir, 2
File with name 'sym_dir3' was created, descriptor index: 6
Directory Listing:
.       => dir, 2
..      => dir, 0
sym_dir3        => sym, 6 -> dir1/dir3
File 'file1' not found.
File 'file1' was opened, File Descriptor: 0.
File with name 'sym_dir2' was created, descriptor index: 7
'dir2/sym_dir3/../sym_dir2' => type=sym nlink=1 size=3 items
'dir2' => type=dir nlink=2 size=3 items
```

Лістинг.

Я використовувала мову C# (.NET). Повний код класів і методів наведений нижче або за посиланням на github.

Коди класів, які реалізують потрібну поведінку:

```
namespace lab4;

public class FileSystem
{
    public bool[] Bitmap { get; set; }
    public List<FileDescriptor?> Descriptors { get; set; }
    public FileDescriptor CurrentDirectory { get; private set; }
    private List<int> FreeFileDescriptorNumbers { get; set; } = new List<int>();
    private List<OpenedFile> OpenedFiles { get; set; } = new List<OpenedFile>();
    private int BlockSize { get; set; } = 64;
    private int MaxSymlinkLoops = 10;
    public Directory RootDirectory { get; set; }

    public FileSystem(int maxNumberOfDescriptors)
    {
        int numberOfBlocks = 20;
        int maxNumberOfNumberDescriptors = 3;
        Bitmap = new bool[numberOfBlocks];
        Descriptors = new List<FileDescriptor?>(maxNumberOfDescriptors);

        for (int i = 0; i < maxNumberOfDescriptors; i++)
        {
            Descriptors.Add(null);
        }

        for (int i = 0; i < maxNumberOfNumberDescriptors; i++)
        {
            FreeFileDescriptorNumbers.Add(i);
        }

        int rootDescriptorIndex = FindFreeDescriptorIndex();
```

```
        if (rootDescriptorIndex == -1)
        {
            throw new InvalidDataException("Can not create root directory.");
        }

        Descriptors[rootDescriptorIndex] = new FileDescriptor(type: FileType.Dir);
        RootDirectory = Descriptors[rootDescriptorIndex]!.Directory;
        RootDirectory.Entries?.Add(new DirectoryEntry(".", rootDescriptorIndex));
        Descriptors[rootDescriptorIndex]!.HardLinkCount++;
        RootDirectory.Entries?.Add(new DirectoryEntry("..", rootDescriptorIndex));
        Descriptors[rootDescriptorIndex]!.HardLinkCount++;

        CurrentDirectory = Descriptors[rootDescriptorIndex]!;
    }

    private (string parentPath, string itemName) GetParentPathAndItemName(string
path)
    {
        string parentPath, itemName;

        if (path == "/")
        {
            parentPath = "/";
            itemName = "";
        }
        else
        {
            path = path.TrimEnd('/');

            var lastIndex = path.LastIndexOf('/');
            itemName = lastIndex == -1 ? path : path[(lastIndex + 1)..];
            parentPath = lastIndex == -1 ? "" : lastIndex == 0 ? "/" :
path[..lastIndex];
        }

        return (parentPath, itemName);
    }

    public FileDescriptor GetDirectoryDescriptorByPath(string path)
    {
        int symlinkLoopCount = 0;
        FileDescriptor? startDescriptor;

        if (path.StartsWith("/"))
        {
            startDescriptor = Descriptors[0];
            path = path.TrimStart('/');
        }
        else
        {
            startDescriptor = CurrentDirectory;
        }
```

```
        if (startDescriptor == null)
        {
            throw new InvalidOperationException("Something wrong occured while
finding root or current directory.");
        }

        if (string.IsNullOrEmpty(path))
        {
            return startDescriptor;
        }

        string[] parts = path.Split(new char[] { '/' },
StringSplitOptions.RemoveEmptyEntries);
        FileDescriptor current = startDescriptor;

        foreach (var part in parts)
        {
            int index = current.Directory.FindDescriptorIndex(part);
            if (index == -1)
            {
                throw new FileNotFoundException($"Part '{part}' of the path was not
found.");
            }

            current = Descriptors[index]!;

            if (current.Type == FileType.Sym)
            {
                if (++symlinkLoopCount > MaxSymlinkLoops)
                {
                    throw new InvalidOperationException("Too many levels of
symbolic links.");
                }

                current = ResolveSymlink(current);
            }

            if (current.Type != FileType.Dir)
            {
                throw new InvalidOperationException("Path is not a directory.");
            }
        }

        return current;
    }

    private FileDescriptor ResolveSymlink(FileDescriptor symlinkDescriptor)
    {
        string targetPath = symlinkDescriptor.SymLinkTarget;

        return GetDirectoryDescriptorByPath(targetPath);
    }
```

```
    private FileDescriptor GetFileDescriptorByIndex(int descriptorIndex, bool
shouldBeReg = true)
    {
        FileDescriptor fileDescriptor = Descriptors[descriptorIndex]!;
        if (fileDescriptor.Type == FileType.Sym)
        {
            var targetPath = fileDescriptor.SymLinkTarget;
            var (targetParentPath, targetFilename) =
GetParentPathAndItemName(targetPath);
            var targetParrentDirectoryDescriptor =
GetDirectoryDescriptorByPath(targetParentPath);

            int targetDescriptorIndex =
targetParrentDirectoryDescriptor.Directory.FindDescriptorIndex(targetFilename);
            fileDescriptor = Descriptors[targetDescriptorIndex]!;
        }

        if (shouldBeReg)
        {
            if (fileDescriptor.Type != FileType.Reg)
            {
                throw new InvalidOperationException($"The path you provided is not
a regular file.");
            }
        }
        else
        {
            if (fileDescriptor.Type != FileType.Dir)
            {
                throw new InvalidOperationException($"The path you provided is not
a directory.");
            }
        }

        return fileDescriptor;
    }

    public void ChangeDirectory(string path)
    {
        var newDir = GetDirectoryDescriptorByPath(path);
        if (newDir.Type != FileType.Dir)
        {
            throw new InvalidOperationException("Target is not a directory.");
        }
        CurrentDirectory = newDir;

        Console.WriteLine($"CWD was changed to '{path}'.");
    }

    public void MakeDirectory(string path)
    {
        var (parentPath, newDirName) = GetParentPathAndItemName(path);
```

```
        if (string.IsNullOrWhiteSpace(newDirName))
        {
            throw new ArgumentException("Directory name cannot be empty.");
        }

        FileDescriptor parentDirDescriptor =
GetDirectoryDescriptorByPath(parentPath);

        if (parentDirDescriptor.Directory.Contains(newDirName))
        {
            Console.WriteLine($"File '{newDirName}' already exists in the
directory.");
            return;
        }

        int newDirDescriptorIndex = FindFreeDescriptorIndex();
        if (newDirDescriptorIndex == -1)
        {
            throw new InvalidOperationException("No free file descriptors
available.");
        }

        FileDescriptor newDirDescriptor = new FileDescriptor(type: FileType.Dir);
        Descriptors[newDirDescriptorIndex] = newDirDescriptor;

        newDirDescriptor.Directory.Entries?.Add(new DirectoryEntry(".",
newDirDescriptorIndex));
        newDirDescriptor.HardLinkCount++;

        int parentDescriptorIndex = FindDescriptorIndex(parentPath);
        newDirDescriptor.Directory.Entries?.Add(new DirectoryEntry("..",
parentDescriptorIndex));
        Descriptors[parentDescriptorIndex]!.HardLinkCount++;

        parentDirDescriptor.Directory.AddEntry(newDirName, newDirDescriptorIndex);
    }

    public void RemoveDirectory(string path)
    {
        var (parentPath, dirName) = GetParentPathAndItemName(path);

        if (string.IsNullOrWhiteSpace(dirName))
        {
            throw new ArgumentException("Directory name cannot be empty.");
        }

        if (dirName == "." || dirName == "..")
        {
            Console.WriteLine($"You can not remove '{dirName}' directory.");
            return;
        }
```

```
        FileDescriptor parrentDirectoryDescriptor =
GetDirectoryDescriptorByPath(parentPath);

        if (!parrentDirectoryDescriptor.Directory.Contains(dirName))
        {
            Console.WriteLine($"Directory '{dirName}' does not exist in the
'{parentPath}'.");
            return;
        }

        var directoryEntry =
parrentDirectoryDescriptor.Directory.Entries.First(entry => entry.FileName ==
dirName);

        int fileDescriptorIndex = directoryEntry.FileDescriptorIndex;
        var fileDescriptor = Descriptors[fileDescriptorIndex]!;
        if (fileDescriptor.Type != FileType.Dir)
        {
            Console.WriteLine($"'{path}' is not a directory.");
            return;
        }

        if (fileDescriptor.Directory.Entries.Count != 2)
        {
            Console.WriteLine($"Directory '{path}' is not empty. You can not remove
it.");
            return;
        }

        parrentDirectoryDescriptor.Directory.Entries?.Remove(directoryEntry);
        parrentDirectoryDescriptor.HardLinkCount--;
        FreeFile(fileDescriptorIndex);

        Console.WriteLine($"Successfully removed directory with pathname
'{path}'.");
    }

    public void CreateSymlink(string targetPath, string linkName)
    {
        var (linkParentPath, linkFileName) = GetParentPathAndItemName(linkName);

        FileDescriptor linkParentDescriptor =
GetDirectoryDescriptorByPath(linkParentPath);
        if (linkParentDescriptor.Directory.Contains(linkFileName))
        {
            Console.WriteLine("Link name already exists.");
            return;
        }

        FileDescriptor symlinkDescriptor = new(type: FileType.Sym)
        {
            SymLinkTarget = targetPath
        };
```

```
        int symlinkDescriptorIndex = FindFreeDescriptorIndex();
        if (symlinkDescriptorIndex == -1)
        {
            Console.WriteLine("No free file descriptors available.");
            return;
        }
        Descriptors[symlinkDescriptorIndex] = symlinkDescriptor;
        linkParentDescriptor.Directory.AddEntry(linkFileName,
symlinkDescriptorIndex);
    }

    private int FindDescriptorIndex(string path)
    {
        if (string.IsNullOrEmpty(path))
        {
            return 0;
        }

        FileDescriptor descriptor = GetDirectoryDescriptorByPath(path);
        return Descriptors.IndexOf(descriptor);
    }

    public bool Create(string path)
    {
        var (directoryPath, fileName) = GetParentPathAndItemName(path);

        var parrentDirectoryDescriptor =
GetDirectoryDescriptorByPath(directoryPath);
        if (parrentDirectoryDescriptor.Type != FileType.Dir)
        {
            throw new InvalidOperationException("Path is not a directory.");
        }

        if (parrentDirectoryDescriptor.Directory.Contains(fileName))
        {
            Console.WriteLine($"File '{fileName}' already exists in the
directory.");
            return false;
        }

        int descriptorIndex = FindFreeDescriptorIndex();
        if (descriptorIndex == -1)
        {
            Console.WriteLine("No free descriptors available.");
            return false;
        }

        var fileDescriptor = new FileDescriptor();
        Descriptors[descriptorIndex] = fileDescriptor;

        parrentDirectoryDescriptor.Directory.AddEntry(fileName, descriptorIndex);
```

```
        return true;
    }

    public void Stat(string path = "")
    {
        var (parentPath, itemName) = GetParentPathAndItemName(path);
        var parrentDirectoryDescriptor = GetDirectoryDescriptorByPath(parentPath);

        int descriptorIndex =
parrentDirectoryDescriptor.Directory.FindDescriptorIndex(itemName);
        if (string.IsNullOrEmpty(itemName))
        {
            if (parentPath == "/")
            {
                descriptorIndex = 0;
            }
            else
            {
                descriptorIndex =
parrentDirectoryDescriptor.Directory.FindDescriptorIndex(".");
            }
        }

        if (descriptorIndex == -1)
        {
            Console.WriteLine($"File '{path}' not found.");
            return;
        }

        FileDescriptor fileDescriptor = Descriptors[descriptorIndex]!;

        Console.Write($"'{path}' =>");
        Console.Write($" type={fileDescriptor.Type.ToString().ToLower()}");
        Console.Write($" nlink={fileDescriptor.HardLinkCount}");

        if (fileDescriptor.Type == FileType.Sym)
        {
            try
            {
                fileDescriptor = GetFileDescriptorByIndex(descriptorIndex,
shouldBeReg: false);
            }
            catch (Exception)
            {
                try
                {
                    fileDescriptor = GetFileDescriptorByIndex(descriptorIndex);
                }
                catch (Exception)
                {
                    Console.WriteLine("");
                }
            }
```

```
        }

        if (fileDescriptor.Type == FileType.Dir)
        {
            Console.WriteLine($" size={fileDescriptor.Directory.Entries.Count}
items");
        }
        else if (fileDescriptor.Type == FileType.Reg)
        {
            Console.Write($", size={fileDescriptor.FileSize}");
            Console.WriteLine($", nblock={fileDescriptor.BlockMap.Count}");
        }
    }

    public void Ls(string path = "")
    {
        var directoryDescriptor = GetDirectoryDescriptorByPath(path);

        Console.WriteLine("Directory Listing:");
        foreach (var entry in directoryDescriptor.Directory.Entries)
        {
            int descriptorIndex =
directoryDescriptor.Directory.FindDescriptorIndex(entry.FileName);
            var fileDescriptor = Descriptors[descriptorIndex]!;
            var type = fileDescriptor.Type;
            Console.WriteLine($"{entry.FileName}\t=> {type.ToString().ToLower()},
{descriptorIndex}{(type == FileType.Sym ? $" -> {fileDescriptor.SymLinkTarget}" :
"")}");
        }
    }

    public int Open(string path)
    {
        var (parentPath, filename) = GetParentPathAndItemName(path);
        var parrentDirectoryDescriptor = GetDirectoryDescriptorByPath(parentPath);

        int descriptorIndex =
parrentDirectoryDescriptor.Directory.FindDescriptorIndex(filename);

        if (descriptorIndex != -1)
        {
            FileDescriptor fileDescriptor =
GetFileDescriptorByIndex(descriptorIndex);

            int fileDescriptorNumber;
            if (FreeFileDescriptorNumbers.Count != 0)
            {
                fileDescriptorNumber = FreeFileDescriptorNumbers.Min();
                FreeFileDescriptorNumbers.Remove(fileDescriptorNumber);
            }
            else
            {
```

```
                Console.WriteLine($"You can not open new file, all number
descriptor is using now.");
                return -1;
            }

            OpenedFiles.Add(new OpenedFile
            {
                FileDescriptorNumber = fileDescriptorNumber,
                DescriptorIndex = descriptorIndex,
                Position = 0,
            });

            Console.WriteLine($"File '{filename}' was opened, File Descriptor:
{fileDescriptorNumber}.");

            return fileDescriptorNumber;
        }
        else
        {
            Console.WriteLine($"File '{filename}' not found.");
            return -1;
        }
    }

    public void Close(int fileDescriptorNumber)
    {
        var openedFile = OpenedFiles.FirstOrDefault(f => f.FileDescriptorNumber ==
fileDescriptorNumber);

        if (openedFile != null)
        {
            FreeFileDescriptorNumbers.Add(fileDescriptorNumber);
            OpenedFiles.Remove(openedFile);

            Console.WriteLine($"File was closed, now File Descriptor
{fileDescriptorNumber} is free.");

            var fileDescriptorIndex = openedFile.DescriptorIndex;
            if (Descriptors[fileDescriptorIndex]!.HardLinkCount == 0 &&
!IsFileOpened(fileDescriptorIndex))
                FreeFile(fileDescriptorIndex);
        }
        else
        {
            Console.WriteLine($"File with descriptor number
'{fileDescriptorNumber}' not found or already closed.");
        }
    }

    public void Seek(int fileDescriptorNumber, int offset)
    {
        var openedFile = OpenedFiles.FirstOrDefault(f => f.FileDescriptorNumber ==
fileDescriptorNumber);
```

```
        if (openedFile != null)
        {
            FileDescriptor fileDescriptor =
GetFileDescriptorByIndex(openedFile.DescriptorIndex);
            if (offset >= 0 && offset < fileDescriptor.FileSize)
            {
                openedFile.Position = offset;
            }
            else
            {
                Console.WriteLine($"Invalid offset for file with descriptor number
'{fileDescriptorNumber}'. Offset must be between 0 and file size.");
            }
        }
        else
        {
            Console.WriteLine($"File with descriptor number
'{fileDescriptorNumber}' not found or not opened.");
        }
    }

    public byte?[] Read(int fileDescriptorNumber, int size)
    {
        var openedFile = OpenedFiles.FirstOrDefault(f => f.FileDescriptorNumber ==
fileDescriptorNumber);

        if (openedFile != null)
        {
            FileDescriptor fileDescriptor =
GetFileDescriptorByIndex(openedFile.DescriptorIndex);

            int remainingBytes = fileDescriptor.FileSize - openedFile.Position;

            if (size > remainingBytes)
            {
                Console.WriteLine($"Requested size ({size} bytes) exceeds the
remaining bytes available for reading ({remainingBytes} bytes).");
                size = remainingBytes;
            }

            byte?[] data = new byte?[size];
            for (int i = 0; i < size; i++)
            {
                int blockIndex = openedFile.Position / BlockSize;
                int blockOffset = openedFile.Position % BlockSize;

                data[i] = fileDescriptor.FileData[blockIndex * BlockSize +
blockOffset];

                openedFile.Position++;
            }
```

```
            return data;
        }
        else
        {
            Console.WriteLine($"File with descriptor number
'{fileDescriptorNumber}' not found or not opened.");
            return Array.Empty<byte?>();
        }
    }

    public void Write(int fileDescriptorNumber, byte[] data)
    {
        var openedFile = OpenedFiles.FirstOrDefault(f => f.FileDescriptorNumber ==
fileDescriptorNumber);

        if (openedFile != null)
        {
            FileDescriptor fileDescriptor =
GetFileDescriptorByIndex(openedFile.DescriptorIndex);

            int remainingBytes = fileDescriptor.FileSize - openedFile.Position;

            if (data.Length > remainingBytes)
            {
                Console.WriteLine($"Not enough space available for writing
{data.Length} bytes.");
                return;
            }

            for (int i = 0; i < data.Length; i++)
            {
                int blockIndex = openedFile.Position / BlockSize;
                int blockOffset = openedFile.Position % BlockSize;
                int byteIndex = blockIndex * BlockSize + blockOffset;

                if (IsBlockConsistsOnlyOfNulls(fileDescriptor.FileData,
blockIndex))
                {
                    fileDescriptor.BlockMap.Add(FindFreeBlockIndex());
                }

                fileDescriptor.FileData[byteIndex] = data[i];

                openedFile.Position++;
            }

            Console.WriteLine($"Successfully wrote {data.Length} bytes to the file
with descriptor number '{fileDescriptorNumber}'.");
        }
        else
        {
            Console.WriteLine($"File with descriptor number
'{fileDescriptorNumber}' not found or not opened.");
```

```
        }
    }

    public void Link(string path1, string path2)
    {
        var (parentPath1, filename1) = GetParentPathAndItemName(path1);
        var parrentDirectoryDescriptor1 =
GetDirectoryDescriptorByPath(parentPath1);

        var (parentPath2, filename2) = GetParentPathAndItemName(path2);
        var parrentDirectoryDescriptor2 =
GetDirectoryDescriptorByPath(parentPath2);

        int fileDescriptorIndex1 =
parrentDirectoryDescriptor1.Directory.Entries?.FirstOrDefault(entry =>
entry.FileName == filename1)?.FileDescriptorIndex ?? -1;

        if (fileDescriptorIndex1 != -1)
        {
            FileDescriptor fileDescriptor =
GetFileDescriptorByIndex(fileDescriptorIndex1);

            parrentDirectoryDescriptor2.Directory.Entries?.Add(new
DirectoryEntry(filename2, fileDescriptorIndex1));
            Descriptors[fileDescriptorIndex1]!.HardLinkCount++;

            Console.WriteLine($"Successfully created a hard link '{path2}' pointing
to the same file as '{path1}'.");
        }
        else
        {
            Console.WriteLine($"File with pathname '{path1}' not found in the
directory.");
        }
    }

    public void Unlink(string path)
    {
        var (parentPath, filename) = GetParentPathAndItemName(path);
        var parrentDirectoryDescriptor = GetDirectoryDescriptorByPath(parentPath);

        var directoryEntry =
parrentDirectoryDescriptor.Directory.Entries?.FirstOrDefault(entry =>
entry.FileName == filename);

        if (directoryEntry != null)
        {
            int fileDescriptorIndex = directoryEntry.FileDescriptorIndex;
            FileDescriptor fileDescriptor =
GetFileDescriptorByIndex(fileDescriptorIndex);

            Descriptors[fileDescriptorIndex]!.HardLinkCount--;
```

```
            Console.WriteLine($"Successfully unlinked the hard link with pathname
'{path}'.");

            parrentDirectoryDescriptor.Directory.Entries?.Remove(directoryEntry);

            if (Descriptors[fileDescriptorIndex]!.HardLinkCount == 0 &&
!IsFileOpened(fileDescriptorIndex))
                FreeFile(fileDescriptorIndex);
        }
        else
        {
            Console.WriteLine($"Hard link with pathname '{path}' not found in the
directory.");
        }
    }

    public void Truncate(string path, int newSize)
    {
        var (parentPath, filename) = GetParentPathAndItemName(path);
        var parrentDirectoryDescriptor = GetDirectoryDescriptorByPath(parentPath);

        int descriptorIndex =
parrentDirectoryDescriptor.Directory.FindDescriptorIndex(filename);
        if (descriptorIndex != -1)
        {
            FileDescriptor fileDescriptor =
GetFileDescriptorByIndex(descriptorIndex);

            var isFileReduced = newSize < fileDescriptor.FileSize;
            if (isFileReduced)
            {
                int oldMaxIndexOfBlock = fileDescriptor.FileSize / BlockSize;
                int newMaxIndexOfBlock = newSize / BlockSize;
                int numberOfReleasedBlocks = oldMaxIndexOfBlock -
newMaxIndexOfBlock;

                for (int i = 0; i < numberOfReleasedBlocks; i++)
                {
                    if (!IsBlockConsistsOnlyOfNulls(fileDescriptor.FileData,
oldMaxIndexOfBlock - i))
                        fileDescriptor.BlockMap.RemoveAt(0);
                }

                fileDescriptor.FileData =
fileDescriptor.FileData.Take(newSize).ToList();
            }
            else
            {
                fileDescriptor.FileData.Capacity = newSize;

                for (int i = fileDescriptor.FileData.Count; i < newSize; i++)
                {
                    fileDescriptor.FileData.Add(null);
```

```
                }
            }

            fileDescriptor.FileSize = newSize;

            Console.WriteLine($"FileSize of '{filename}' was {(isFileReduced ?
"reduced" : "increased")} to {newSize} bytes.");
        }
        else
        {
            Console.WriteLine($"File '{filename}' not found.");
        }
    }

    private bool IsBlockConsistsOnlyOfNulls(List<byte?> data, int blockIndex)
    {
        int startIndex = blockIndex * BlockSize;
        int endIndex = startIndex + BlockSize - 1;

        if (endIndex > data.Count - 1)
            endIndex = data.Count - 1;

        for (int i = startIndex; i <= endIndex; i++)
        {
            if (data[i] != null)
            {
                return false;
            }
        }

        return true;
    }

    private int FindFreeBlockIndex()
    {
        for (int i = 0; i < Bitmap.Length; i++)
        {
            if (!Bitmap[i])
            {
                Bitmap[i] = true;
                return i;
            }
        }
        return -1;
    }

    private int FindFreeDescriptorIndex()
    {
        for (int i = 0; i < Descriptors.Count; i++)
        {
            if (Descriptors[i] == null)
            {
                return i;
```

```
            }
        }
        return -1;
    }

    private void FreeFile(int fileDescriptorIndex)
    {
        if (fileDescriptorIndex >= 0 && fileDescriptorIndex < Descriptors.Count)
        {
            foreach (int blockIndex in Descriptors[fileDescriptorIndex]!.BlockMap)
            {
                Bitmap[blockIndex] = false;
            }

            Descriptors[fileDescriptorIndex] = null;

            Console.WriteLine($"Successfully released file descriptor index
'{fileDescriptorIndex}'.");
        }
        else
        {
            Console.WriteLine($"Invalid file descriptor index
'{fileDescriptorIndex}'.");
        }
    }

    private bool IsFileOpened(int fileDescriptorIndex)
    {
        return OpenedFiles.Any(openedFile => openedFile.DescriptorIndex ==
fileDescriptorIndex);
    }
}
```

```
namespace lab4;

public enum FileType { Reg, Dir, Sym }

public class FileDescriptor
{
    public FileType Type { get; set; }
    public int HardLinkCount { get; set; } = 1;
    public int FileSize { get; set; } = 0; // only for regular files
    public List<byte?> FileData { get; set; } = new List<byte?>(); // only for
regular files
    public List<int> BlockMap { get; set; } = new List<int>(); // only for regular
files
    public Directory Directory { get; set; } = new(); // only for directories
    public string SymLinkTarget { get; set; } = ""; // only for symbolic links

    public FileDescriptor(FileType type = FileType.Reg)
    {
```

```
        Type = type;

        if (Type == FileType.Dir)
        {
            Directory = new Directory();
        }
    }
}
```

```
namespace lab4;

public class Directory
{
    public List<DirectoryEntry> Entries { get; } = new List<DirectoryEntry>();

    public bool Contains(string fileName)
    {
        return Entries.Any(entry => entry.FileName == fileName);
    }

    public void AddEntry(string fileName, int fileDescriptorIndex)
    {
        Entries.Add(new DirectoryEntry(fileName, fileDescriptorIndex));
        Console.WriteLine($"File with name '{fileName}' was created, descriptor
index: {fileDescriptorIndex}");
    }

    public int FindDescriptorIndex(string filename)
    {
        for (int i = 0; i < Entries.Count; i++)
        {
            if (Entries[i].FileName == filename)
            {
                return Entries[i].FileDescriptorIndex;
            }
        }
        return -1;
    }
}
```

```
namespace lab4;

public class DirectoryEntry
{
    public string FileName { get; set; }
    public int FileDescriptorIndex { get; set; }

    public DirectoryEntry(string fileName, int fileDescriptorIndex)
```

```
    {
        FileName = fileName;
        FileDescriptorIndex = fileDescriptorIndex;
    }
}
```

```
namespace lab4;

public class OpenedFile
{
    public int FileDescriptorNumber { get; set; }
    public int DescriptorIndex { get; set; }
    public int Position { get; set; }
}
```