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 №4

«Файлова система (частина 1)»

з дисципліни
«Операційні системи»

Виконала:

студентка групи ІМ-11
Бащак Ярина Володимирівна

Перевірив:

ст. вик. Сімоненко А.В.

Київ 2023

Завдання.

Завдання на роботу

Розробити драйвер ФС та частину підсистеми ядра, яка підтримує файлові системи, що відповідає наведеній вище семантиці ФС. Драйвер ФС та частину ядра реалізувати в програмі користувача. Використовуйте наведену вище ідею структури дескриптора файлу. Використайте вміст звичайного файлу в наявній ФС в якості вмісту носія інформації, розмір файлу визначає розмір носія інформації. Самостійно вибрати розмір блоку, розмір мапи номерів блоків у дескрипторі файлу, максимальну довжину імені файлу. Розроблена програма не має бути ФС типу bypass чи pass through, усі дії з ФС мають відбуватися в розробленій програмі. Використовувати FUSE або схожу систему не дозволяється, оскільки в лабораторній роботі треба реалізувати частину підсистеми ядра. Якщо використовувати FUSE, тоді буде використовуватися наявна підсистема ядра.

Замість використання вмісту звичайного файлу в наявній ФС в якості вмісту носія інформації, можна використати пам’ять як носій інформації. Тобто можна розробити драйвер ФС для пам’яті. У цьому випадку бітова карта не потрібна. Для представлення вмісту директорії можна використати щось більш оптимальніше ніж дані в блоках. Замість мапи номерів блоків можна використати будь-яку структуру даних, яка дозволить ефективно кодувати номери блоків та блоки, вміст яких складається з нулів. Кількість дескрипторів файлів та кількість блоків визначати не треба. Вміст звичайного файлу все одно повинен складатися з блоків.

Розробити програму, яка підтримує наступні команди (замість введення команд можна викликати необхідні функції безпосередньо в програмі):

- `mkfs n` – ініціалізувати ФС, `n` – це кількість дескрипторів файлів (ця команда не потрібна для ФС для пам’яті).
- `stat name` – вивести інформацію про файл (дані дескриптору файлу).
- `ls` – вивести список жорстких посилань на файли з номерами дескрипторів файлів в директорії.
- `create name` – створити звичайний файл та створити на нього жорстке посилання з ім’ям `name` у директорії.
- `fd = open name` – відкрити звичайний файл, на який вказує жорстке посилання з ім’ям `name`. Команда повинна призначити найменше вільне невід’ємне цілочисельне значення (назвемо його *числовий дескриптор файлу*) для роботи з відкритим файлом (це число – це не те саме, що номер дескриптору файлу у ФС). Один файл може бути відкритий кілька разів. Кількість числових дескрипторів файлів може бути обмежена.
- `close fd` – закрити раніше відкритий файл з числовим дескриптором файлу `fd`, значення `fd` стає вільним.
- `seek fd offset` – вказати зміщення для відкритого файлу, де почнеться наступне читання або запис (далі «зміщення»). При відкритті файлу зміщення дорівнює нулю. Це зміщення вказується тільки для цього `fd`.
- `read fd size` – прочитати `size` байт даних з відкритого файлу, до значення зміщення додається `size`.
- `write fd size` – записати `size` байт даних у відкритий файл, до значення зміщення додається `size`.
- `link name1 name2` – створити жорстке посилання з ім’ям `name2` на файл, на який вказує жорстке посилання з ім’ям `name1`.
- `unlink name` – знищити жорстке посилання з ім’ям `name`.

- `truncate name size` – змінити розмір файлу, на який вказує жорстке посилання з ім’ям `name`. Якщо розмір файлу збільшується, тоді неініціалізовані дані дорівнюють нулям.

Драйвер ФС звільнює дані файлу, якщо жодне жорстке посилання не вказує на файл і файл не є відкритим. На файл може не вказувати жодне жорстке посилання, але якщо він відкритий, тоді з ним можна виконувати команди команди `seek`, `read` та `write`.

Якщо команда `truncate` збільшує розмір файлу, тоді реалізувати оптимізацію та не створювати блоки, вміст яких складається з нулів.

Приклади виконання всіх команд.

- ініціалізація ФС. За умовою є одна директорія, тому при створенні ФС також створюється дескриптор цієї директорії.

```
static void Main(string[] args)
{
    int maxNumberOfDescriptors = 5;
    var fileSystem = new FileSystem(maxNumberOfDescriptors);
```

```
$ dotnet run
File descriptor for directory was created.
```

- create name, ls. Більше 5 дескрипторів не доступно

```
        fileSystem.Create("file1.txt");
        fileSystem.Create("file2.txt");
        fileSystem.Create("file3.txt");
        fileSystem.Create("file4.txt");
        fileSystem.Create("file5.txt");
        fileSystem.Ls();
```

```
$ dotnet run
File descriptor for directory was created.
File descriptor for file was created.
File with name 'file1.txt' was created, descriptor index: 1
File descriptor for file was created.
File with name 'file2.txt' was created, descriptor index: 2
File descriptor for file was created.
File with name 'file3.txt' was created, descriptor index: 3
File descriptor for file was created.
File with name 'file4.txt' was created, descriptor index: 4
No free descriptors available.
Directory Listing:
file1.txt       => reg, 1
file2.txt       => reg, 2
file3.txt       => reg, 3
file4.txt       => reg, 4
```

- open, close. Більше 3 одночасно відкритих файлів не доступно

```
        var fd1 = fileSystem.Open("file2.txt");
        var fd2 = fileSystem.Open("nonExistedFile.txt");
        var fd3 = fileSystem.Open("file1.txt");
        var fd4 = fileSystem.Open("file1.txt");
        var fd5 = fileSystem.Open("file1.txt");
        fileSystem.Close(fd4);
        fileSystem.Close(fd1);
        var fd6 = fileSystem.Open("file1.txt");
        var fd7 = fileSystem.Open("file2.txt");
```

```
File 'file2.txt' was opened, File Descriptor: 0.
File 'nonExistedFile.txt' not found.
File 'file1.txt' was opened, File Descriptor: 1.
File 'file1.txt' was opened, File Descriptor: 2.
You can not open new file, all number descriptor is using now.
File was closed, now File Descriptor 2 is free.
File was closed, now File Descriptor 0 is free.
File 'file1.txt' was opened, File Descriptor: 0.
File 'file2.txt' was opened, File Descriptor: 2.
```

- truncate, write, seek, read

```
fileSystem.Truncate("file2.txt", 124);
byte[] byteArray = new byte[60];

for (int i = 0; i < byteArray.Length; i++)
{
    byteArray[i] = (byte)i;
}
fileSystem.Write(fd1, byteArray);

fileSystem.Seek(fd1, 20);

var data = fileSystem.Read(fd1, 300);
Console.WriteLine("Byte Array:");
foreach (var item in data)
{
    if (item == null)
        Console.Write("\\0 ");
    else
        Console.Write(item + " ");
}
Console.WriteLine();
```

```
File 'file2.txt' was opened, File Descriptor: 0.
FileSize of 'file2.txt' was increased to 124 bytes.
Successfully wrote 60 bytes to the file with descriptor number '0'.
Requested size (300 bytes) exceeds the remaining bytes available for reading (104 bytes).
Byte Array:
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 5
1 52 53 54 55 56 57 58 59 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
\0 \0 \0 \0 \0 \0 \0 \0 \0
```

- stat, trancate. Розмір файлу збільшили до 128 байтів (trancate). Розмір одного блоку: 64 байти. У файл починаючи з 64-го індексу (seek) записали 20 байтів (write), тобто мав зайнятись лише один блок (stat). Потім скоротили файл до розміру 60 байтів (trancate), які є нульовими, тому зайнятих блоків стало 0 (stat).

```
fileSystem.Truncate("file2.txt", 128);
fileSystem.Seek(fd1, 64);

byte[] byteArray = new byte[20];

for (int i = 0; i < byteArray.Length; i++)
{
    byteArray[i] = (byte)i;
}
fileSystem.Write(fd1, byteArray);
fileSystem.Stat("file2.txt");
fileSystem.Truncate("file2.txt", 60);
fileSystem.Stat("file2.txt");
```

```
File 'file2.txt' was opened, File Descriptor: 0.
FileSize of 'file2.txt' was increased to 128 bytes.
Successfully wrote 20 bytes to the file with descriptor number '0'.
'file2.txt' => type=reg, nlink=1, size=128, nblock=1
FileSize of 'file2.txt' was reduced to 60 bytes.
'file2.txt' => type=reg, nlink=1, size=60, nblock=0
```

- link, unlink. Якщо жодного жорсткого посилання не вказує на файл, але він відкритий, то звільнення не відбувається. У файл все ще можна, наприклад, щось записати. І тільки після закриття файлу, звільняються блоки, які займав цей файл.

```
        fileSystem.Link("file2.txt", "file3.txt");
        fileSystem.Stat("file2.txt");
        var fd1 = fileSystem.Open("file2.txt");
        fileSystem.Unlink("file2.txt");
        fileSystem.Unlink("file3.txt");

        byte[] byteArray = new byte[20];

        for (int i = 0; i < byteArray.Length; i++)
        {
            byteArray[i] = (byte)i;
        }
        fileSystem.Write(fd1, byteArray);

        fileSystem.Close(fd1);
```

```
File with name 'file2.txt' was created, descriptor index: 2
FileSize of 'file2.txt' was increased to 128 bytes.
Successfully created a hard link 'file3.txt' pointing to the same file as 'file2.txt'.
'file2.txt' => type=reg, nlink=2, size=128, nblock=0
File 'file2.txt' was opened, File Descriptor: 0.
Successfully unlinked the hard link with name 'file2.txt'.
Successfully unlinked the hard link with name 'file3.txt'.
Successfully wrote 20 bytes to the file with descriptor number '0'.
File was closed, now File Descriptor 0 is free.
Successfully released the file with descriptor index '2'.
```

Лістинг.

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

Коди класів, які реалізують потрібну поведінку:

```
public class FileSystem
{
    public bool[] Bitmap { get; set; }
    public List<FileDescriptor?> Descriptors { get; 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;
```

```
    public Directory Directory { get; set; }

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

        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)
        {
            Descriptors[rootDescriptorIndex] = new FileDescriptor(false);
        }
    }

    public bool Create(string fileName)
    {
        if (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;

        Directory.AddEntry(fileName, descriptorIndex);

        return true;
    }

    public void Stat(string filename)
    {
```

```
        int descriptorIndex = Directory.FindDescriptorIndex(filename);
        if (descriptorIndex != -1)
        {
            FileDescriptor fileDescriptor = Descriptors[descriptorIndex]!;
            Console.Write($"'{filename}' =>");
            Console.Write($" type={(fileDescriptor.IsRegularFile ? "reg" :
"dir")}");
            Console.Write($", nlink={fileDescriptor.HardLinkCount}");
            Console.Write($", size={fileDescriptor.FileSize}");
            Console.WriteLine($", nblock={fileDescriptor.BlockMap.Count}");
        }
        else
        {
            Console.WriteLine($"File '{filename}' not found.");
        }
    }

    public void Ls()
    {
        Directory.Ls();
    }

    public int Open(string filename)
    {
        int descriptorIndex = Directory.FindDescriptorIndex(filename);
        if (descriptorIndex != -1)
        {
            FileDescriptor fileDescriptor = Descriptors[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 =
Descriptors[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 =
Descriptors[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 =
Descriptors[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 name1, string name2)
    {
        int fileDescriptorIndex1 = Directory.Entries?.FirstOrDefault(entry =>
entry.FileName == name1)?.FileDescriptorIndex ?? -1;

        if (fileDescriptorIndex1 != -1)
        {
            Directory.Entries?.Add(new DirectoryEntry(name2,
fileDescriptorIndex1));
            Descriptors[fileDescriptorIndex1]!.HardLinkCount++;

            Console.WriteLine($"Successfully created a hard link '{name2}' pointing
to the same file as '{name1}'.");
```

```
        }
        else
        {
            Console.WriteLine($"File with name '{name1}' not found in the
directory.");
        }
    }

    public void Unlink(string name)
    {
        var directoryEntry = Directory.Entries?.FirstOrDefault(entry =>
entry.FileName == name);

        if (directoryEntry != null)
        {
            int fileDescriptorIndex = directoryEntry.FileDescriptorIndex;
            Descriptors[fileDescriptorIndex]!.HardLinkCount--;

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

            Directory.Entries?.Remove(directoryEntry);

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

    public void Truncate(string filename, int newSize)
    {
        int descriptorIndex = Directory.FindDescriptorIndex(filename);
        if (descriptorIndex != -1)
        {
            FileDescriptor fileDescriptor = Descriptors[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($"Successfully released the file with descriptor
index '{fileDescriptorIndex}'.");
        }
        else
        {
            Console.WriteLine($"Invalid file descriptor index
'{fileDescriptorIndex}'.");
        }
    }

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

public class FileDescriptor
{
    public bool IsRegularFile { get; set; }
    public int HardLinkCount { get; set; } = 1;
    public int FileSize { get; set; } = 0;
    public List<byte?> FileData { get; set; } = new List<byte?>();
    public List<int> BlockMap { get; set; } = new List<int>();
```

```
    public FileDescriptor(bool isRegularFile = true)
    {
        IsRegularFile = isRegularFile;

        Console.WriteLine($"File descriptor for {(IsRegularFile ? "file" :
"directory")} was created.");
    }
}

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;
    }

    public void Ls()
    {
        Console.WriteLine("Directory Listing:");
        foreach (var entry in Entries)
        {
            Console.WriteLine($"{entry.FileName}\t=> reg,
{entry.FileDescriptorIndex}");
        }
    }
}

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

```
    public DirectoryEntry(string fileName, int fileDescriptorIndex)
    {
        FileName = fileName;
        FileDescriptorIndex = fileDescriptorIndex;
    }
}

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