Operating Systems 2024
Assignment 3: File System Implementation

Deadline: Friday, May 24, 2024, 18:00

1 Introduction

A disk can be accessed as an array of disk blocks. Typical disk block sizes are 512, 1024, 2048, 4096 or 8192 bytes. In general, larger file systems employ larger disk blocks in order to keep things manageable. In order to store files and directories in such an array of blocks, we need to think about how to organize the data. In which of the free blocks will we write a given file? Which blocks on the device are actually free (not in use)? How can we find out in which blocks the file's content is located? How do we store other data about a file, such as its permissions and time of last modification? Finally, how do we store directories? Several "formats" to organize such data have been devised over the years, such as FAT, NTFS, ext2, HFS and XFS. We usually refer to these formats as *file systems*.

A file system can be split in roughly two parts: the actual data and the metadata about this data. The metadata contains a table of filenames and information about these filenames such as permissions, file size and more importantly a list of disk blocks where the contents of the file can be found. Moreover, the directory structure (tree) must be stored as well!

In this final assignment we will implement parts of a simple file system, named EdFS[1]. The initial code that you are provided with only provides a framework and does not read any data from the file system. The technical design of the file system is described in Section 5 of the separate appendices document. You will work on the implementation of support to read directories, to make/remove directories, to read files and finally to create and write files. The tasks gradually increase in complexity. For the basic tasks some guidance is included in the appendices (separate file). There is less guidance for the more difficult parts and in order to be awarded an excellent grade we expect you to be able to work out solutions on your own.

Usually implementations of file systems are done as part of an operating system, for example as kernel module. However, for this assignment we will be using the FUSE system[2]. FUSE (Filesystem in User SpacE) allows file systems to be implemented in user space. The FUSE infrastructure will handle all necessary communication with the kernel to make this possible. Please refer to the appendices (separate file) for more information on how to use FUSE.

To facilitate development we will not be storing the file system on an actual device, but within a file. So, in fact a file on the host computer is assuming the role of disk. We will refer to this file as an *image file*. A populated image file with file content and an empty initialized image file are available from Brightspace and should be used to test your implementation.

[1] Educational File System
[2] `https://github.com/libfuse/libfuse`

2 Requirements

Although the starting point you are given is written in pure C, it is allowed to use C++. We expect the following to be achieved:

- Implement support for reading (sub)directories such that the full directory tree stored on the provided reference image can be read.
- Implement support for reading files from the file system. Your implementation must be able to read all files on the provided reference file system image. It must be able to cope with reads of arbitrary size and at arbitrary offsets. We will verify this by computing the checksums of the read files and comparing these with the reference.
- Implement support for the creation and removal of (sub)directories. The user must be able to create/remove directories using *mkdir* and *rmdir*.
- Implement support for the creation of new files. This should correctly create new inodes and directory entries.
- Implement support for writing to files.
  - When necessary, new disk blocks must be allocated and be registered in the inode for the corresponding file.
  - The file size must be correctly updated in the inode. Changes must be persistent after re-mounting the file system.
  - We have provided a utility called *overwrite* which overwrites a given file with a specific pattern. After using this utility, it must be possible to read back the correct contents of the file (for example using *cat*) and *md5sum* must be able to compute the correct checksum before and after re-mounting the filesystem. See also the section on Testing below.
  - The *overwrite* utility must work on files in both the root directory as well as subdirectories.
- Implement support for file truncation, so it becomes possible to fully overwrite files.
- For all changes made to the file system holds:
  - Files and directories must be stored on disk according to the specification.
  - Changes must be persistent after re-mounting the file system.
  - *fsck.edfs* must always pass (report no errors) when run on an unmounted file system, also after the file system has been modified.
- The code that you will write must be modular and avoid extensive code duplication. This will be assessed and reflected in the Quality component of your grade.
- *Important!* Make sure to use extensive error handling in your code so that your code detects various kinds of failures. See below for a list of commonly used error codes.

3 Submission and Grading

You may work in teams of at most *two* persons. The **deadline** is Friday, May 24, 2024, 18:00. Submit your assignments according to the instructions below.

The maximum grade that can be obtained is 10. The grade is the sum of the scores for the following components, maximum score between square brackets:

- [**1.5 out of 10**] Code layout and quality
- [**1.5 out of 10**] Reading directories
- [**2.0 out of 10**] Reading files
- [**2.0 out of 10**] Creating/removing directories
- [**2.0 out of 10**] Writing to (existing) files
- [**0.5 out of 10**] Creating files
- [**0.5 out of 10**] Truncating files

For *code quality* the following is considered: good structure and modularity, consistency of the indentation and brace style, comments where these are required, quality of error handling.

The following needs to be submitted:

- The source code of the FUSE program with your modifications. *Make sure that all files that you have modified contain your names and student IDs.*

What does *not* have to be submitted: any object files, binaries and in particular image files. Please **remove** these from your source code directory before handing in, to keep the tar files as small as possible.

Give your file the following name:

`sXXXXXXX-sYYYYYYY-lab3.tar.gz`

Substitute `XXXXXXX` and `YYYYYYY` with your student IDs.
Submit the tar-archive through the Brightspace submission site. Also note your names and student IDs in the text box in the submission website. *Please, ensure only one team member submits the assignment, such that there is a single submission per team!*

Finally, please note the following:

- All submitted source code and reports will be subject to (automatic) **plagiarism checks** using Turnitin, MOSS, or similar. Suspicions of fraud and plagiarism will be reported to the Board of Examiners.
- The use of text or code generated by ChatGPT or other AI tools is **not allowed**. You are required to implement the requested source code **yourself**, and to write the report **yourself**.
- We may always invite teams to elaborate on their submission in an interview in case parts of the source code need further explanation.
- In case you reuse your own work from a previous year, mention this in a README file or within the assignment source code.
- As with all other course work, keep assignment solutions to yourself. Do not post the code on public Git or code snippet repositories where it can be found by other students. If you use Git, make sure your repository is **private**.
- Test on the university Linux computers before handing in. In the case of disputes, the university Linux installation is used as reference.