Operating Systems 2024
Assignment 3 Appendices

1 FUSE

1.1 Using FUSE

To use FUSE on your own computer, make sure to install the `libfuse-dev` package (Debian/Ubuntu). We have tested the assignment on the university computers (Ubuntu 22.04), where FUSE version 2.9.7 is used. We strongly advice against the use of FUSE 3, as there are cases where different calls are made to the functions implemented by you. This means that the expected behavior will deviate from what is described in this document for a FUSE 2 implementation.

If you want to work remotely on the university computers, then you need to enable the following environment before working on the assignment:

```
source /vol/share/groups/liacs/scratch/os2024/os2024.bashrc
```

The starting point can be compiled using the supplied `Makefile`. An initialized file system image can be obtained from Brightspace. To be able to mount this file system, first a mountpoint has to be created. **Important: on the university computers this mountpoint *cannot* be located on a network share, such as your home directory**. What does work is creating a directory under */tmp*, for example */tmp/testmyusername*. Now the filesystem can be mounted by running:

```
./edfuse myimage.img /tmp/testmyusername
```

Enter the mountpoint to browse the contents of the filesystem. To unmount the filesystem use:

```
fusermount -u /tmp/testmyusername
```

To facilitate debugging it helps to provide the `-f` and `-s` options to *edfuse*:

```
./edfuse -f -s myimage.img /tmp/testmyusername
```

This way, you can also run *edfuse* from within a debugger, such as gdb, to debug your code.

1.2 FUSE API

The FUSE API is structured around the concept of a Virtual File System (VFS). Using a VFS, we have an overview of the entire file system of a computer system (which comprises multiple file systems) as well as a generic interface to the functions of the correct file system implementation in order to carry out the desired operation on a file or directory. The interface makes it easy to implement file systems which is essentially done by implementing the functions in the FUSE file operations structure.

In the starting point that is provided to you, you can find the FUSE file operations structure at the bottom of the `edfuse.c` file. As you can see, functions are registered here for the different functionalities of the file system, such as reading directories, writing files and creating directories. This file operations structure is passed to the FUSE library during initialization in the main function.

The function prototypes for the different operations can be found in the FUSE header file, but these that are needed have already been stubbed out for you in the `edfuse.c` source code file. The arguments required for the different operations are straightforward. In most cases the file or directory to be operated on is specified using a `path` string.

Again, please make sure that you are using FUSE 2 instead of FUSE 3, especially if you are working on this assignment on your own machine.

1.3 Assignment Structure

Note that next to `edfuse.c` the starting point contains:

- `edfs.h` general EdFS definitions, in particular struct definitions for the different data structures.
- `edfs-common.[ch]` generic EdFS routines that would be shared with other EdFS utilities.

We recommend that you adhere to this structure. When writing a new function consider whether this function is specific to the FUSE implementation or is more generic. In case of the latter, add it to `edfs-common.[ch]`.

Several functions to help you implement the file system code are already provided, such as functions to read and write inodes, to manipulate the inode table, `edfs_get_parent_inode`, FUSE functions for the `getattr` and `open` calls. The FUSE function for the `readdir` call has been partly implemented and the same holds for `edfs_find_inode`. The implementation of these two functions must be completed by you. Also, do not take these functions for granted. Make sure to study these functions and understand how they can be used.

1.4 Resources

The following resources may be of use when getting up to speed with the FUSE API:

- FUSE API documentation: `http://libfuse.github.io/doxygen/`
- Documentation about the different FUSE operations: `http://libfuse.github.io/doxygen/structfuse__operations.html`
- FUSE tutorial `http://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/`

2 Error Codes

When writing the file system code we ask you to make extensive use of error handling and to return appropriate error codes when errors are detected. Common error codes can be found in `errno.h`, use `man errno`. To help you, the following error codes are the most common:

- `ENOSPC`, no space left on device (file system is full).
- `EINVAL`, invalid value / argument specified, for instance when a given filename is invalid (too long or contains invalid characters).
- `EIO`, I/O error.
- `ENOENT`, entry does not exist.
- `ENOTDIR`, entry is not a directory.
- `EISDIR`, entry is a directory (and not a regular file).
- `ENOTEMPTY`, the directory is not empty.
- `ENOSYS`, the function is not implemented.
- `EEXIST`, entry already exists.
- `EFBIG`, file too large.

Return values that signal failure conditions are typically negative. The above error codes are positive numbers, so usually the negative number is returned: `-EEXIST`.

3 Testing

In this section, we give some guidance on how to test your implementation. In order to do so, we have provided several files and utilities. Note that after modifying the file system (creating/removing directories, writing files), it is important to also unmount the file system and mount it again to verify the made changes are persistent. To verify the file system is not corrupted after making changes to it, you must use the *fsck.edfs* utility. This utility ensures that the data on disk (so actually within the file system image) is correct and not corrupted. It should report no errors. Though, remember that the file system check might not catch every possible error!

The binary executable file for *fsck.edfs* can be obtained from Brightspace or from `/vol/share/groups/liacs/scratch/os2024/prefix/bin`. If your computer runs a Linux distribution from the last 3 years, the binary will most probably work. If not, contact us on the mailing list and we will look into compiling a different binary. We cannot supply the source code. (In case you do not trust binaries from us, run the binaries on a university computer instead or use a virtual machine).

3.1 Reading directories

You will have to start with the implementation of support for reading directories. To test your implementation, use the populated image file provided on Brightspace. You should be able to inspect a hierarchy of subdirectories using *ls*, *cd*, *find* and so on. A textual representation of the hierarchy as stored in the EdFS image is also available on Brightspace.

3.2 Creating and Removing Directories

To test the support for creating and removing (sub)directories, you can use the *mkdir* and *rmdir* utilities. These take the directory to create/remove as an argument. You should test creating subdirectories in the root directory of the EdFS file system as well as creating subdirectories in subdirectories (nesting).

3.3 Reading Files

You can manually test your implementation for reading files using the *cat*, *dd* and *md5sum* utilities. With the *md5sum* command MD5 checksums can be computed. The correct MD5 checksums, and file sizes, for the files present on the populated file system image can be found in the listing available on Brightspace (`populated-listing.txt`).

Note that with these utilities your FUSE module will always get read requests for blocks of 4 KiB. This is due to the kernel's page cache. To be able to test reads of different sizes, you need to circumvent the kernel page cache. This can be achieved using direct I/O. For example, the following command tests reads in blocks of 256 bytes and outputs the checksum:

```
dd if=file1.txt bs=256 iflag=direct | md5sum
```

To further facilitate you in testing reads, we provide a Python script, named `testread.py`, with a set of tests. As command line argument you need to provide the path to the mountpoint of your EdFS file system. You must mount the populated file system provided to you. The script will read different files, using different block sizes, and validate the MD5 checksums.

3.4 Writing Files

For testing file write support a custom *overwrite* utility is provided, which can be found at the same locations as *fsck.edfs*. You need to specify an *existing* file name as an argument. *Be careful: the utility*

will overwrite the specified file without asking for confirmation! The utility will overwrite the specified file with a particular pattern. By default this pattern is 3550 bytes in length (so files that were originally larger than 3550 bytes are only partially overwritten!).

Note that the initialized file system image contains five files named from `file1.txt` to `file5.txt`. These files also contain a specific pattern. When you specify one of these files to *overwrite* (*overwrite* compares the filename), a special action will be performed that is defined for that specific file. Ensure to test your code by using *overwrite* on each of the five test files. After overwriting the file, inspect the new file size and its contents. Also compute the MD5 checksum using the *md5sum* command. The following table specifies the correct checksums (MD5) and size for the ***overwritten*** files:

file1.txt	`f4c5d24f23c0fe539cb09d526b5de899`	1278 bytes
file2.txt	`f6cc4fee814b1b0f035116d33b177ed8`	43310 bytes
file3.txt	`7adcd2bf8b139fe71de82246ff7efcf8`	16330 bytes
file4.txt	`ac66cd60692e44e0f0b5ad9cfeb7db0a`	16330 bytes
file5.txt	`bdebaadb35691a06d35b86bd258ddb54`	15265 bytes

3.5 Creating and Truncating Files

The creation of new files can be tested using the default *echo* and *dd* commands. The *touch* command can be useful to test solely file creation (ignore the FUSE warning on not being able to set times). Finally, file truncation can be tested with, for instance, the regular *dd* and *truncate* command line utilities.

4 Getting Started

To help you get started, we will be giving some guidance on the different subtasks in this section. Read the specification of the file system in Section 5 together with this guidance to fully understand what needs to be done before starting implementation.

4.1 Reading Directories

An inode either describes a file or a directory. In the case of a directory, at most both direct blocks may be allocated. Within these blocks the directory entries are stored. To read directories the implementation of the function `edfuse_readdir` must be completed: at the TODO marker you need to write code that reads the allocated disk blocks and extracts directory entries. For every valid directory entry the filler function must be called.

Additionally, you need to complete the implementation of the `edfs_find_inode` function. At the TODO marker you need to write code to, again, visit all valid directories. So, make sure to write a generic function which visits directories of a given inode! You can then reuse this later (you will need such a function more often).

4.2 Reading Files

Completing the `edfs_find_inode` function, as described in the previous subsection, is a prerequisite for working on the reading of files. In fact, this is the first function you will need to call from `edfuse_read`. You need to find the inode in order to determine whether the given path exists and you need the inode to obtain the block numbers of the data blocks where the file's data is stored.

The following arguments are provided to the read function: `path` which is the path to perform the read operation from, `buf` which is the buffer in which the read data should be stored, `size` is the number

of bytes to read, `offset` specifies the position in the file where the bytes should be read and finally `fi` provides some information about the file (which does not need to be used).

Write a generic function which is able to translate the given file offset to a block number and an offset within this block. First think about how this translation would work (use pen and paper). Once you have a good idea, write the necessary implementation. Make sure to propose appropriate error codes if the offset is out of bounds.

It is important to take block boundaries into account. For example, consider a block size of 512 bytes. We are requested to read 100 bytes, starting at offset 500 of the block. Now, you need to read only 12 bytes from this first block and read the remainder from *another* block. Your implementation of the read function must be able to cope with reads that involve multiple disk blocks.

4.3 Creating and Removing Directories

To add the ability to create and remove (sub)directories, the functions `edfs_mkdir` and `edfs_rmdir` should be implemented. Stub functions have already been created for you and registered in the `fuse_operations` structure.

The *mkdir* function takes two arguments: a path and a mode. Because we do not support file permissions, the mode is ignored. The path indicates the full path to the new directory to be created. This path must be split into the name of the new subdirectory and its parent path (that is, the directory in which the new subdirectory should be created). Make sure to perform all necessary validations, such as whether the given name for the new directory is a valid name within EdFS and whether the parent directory exists and is a directory. Then you can create a new inode for the new directory and register this new inode in the parent directory. For the latter, you need to write a function that creates a new directory entry in the parent directory in which the name of the new directory is stored together with the new inode number. Also make this function for registering directory entries generic, you will need it more often. If the registration of the directory entry succeeds, don't forget to commit the inode by writing it to disk.

To complete this task you can use various utility functions that are already provided: `edfs_get_parent_inode`, `edfs_get_basename`, `edfs_new_inode`, `edfs_write_inode`.

For the new inode you do not have to allocate disk blocks, this should be done on demand. However, this does mean that when registering new entries in the parent directory, you may need to allocate a new disk block for the parent directory in case the current disk block is full or no disk block has yet been allocated. To allocate a block, scan the bitmap for a free (unallocated) block. Determine the block number, mark this block as allocated in the bitmap and finally register this block number in the inode. Don't forget to write the inode to disk. Try to make these functions generic, also these functions will be re-used.

Implementing the *rmdir* function is easier. A single path is given as argument, which is the directory to remove. Look for the correct inode. Verify that the directory to remove is indeed empty. Remove the directory entry from its parent and release all other resources that were allocated for this directory. Finally, don't forget to mark the inode as free in the inode table.

4.4 Creating and Writing Files

To be able to create new files, the `edfs_create` operation must be implemented. This function must create a new inode and register this inode in the parent directory (at the same time you can verify this file does not yet exist). The file size should be set to zero initially and no blocks should be allocated.

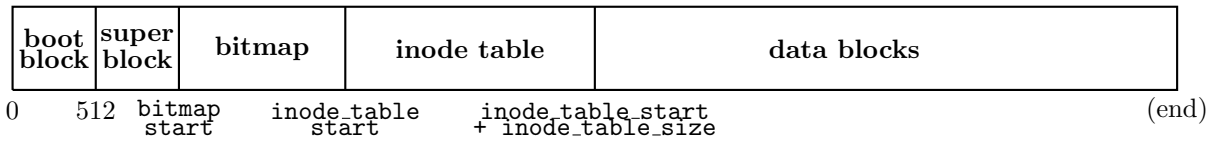
For writing to existing files the `edfs_write` operation is used. This operation is in general similar to the read operation, with two exceptions. Firstly, when necessary new blocks must be allocated. Secondly, if you have modified the file size, the inode must be updated and written to disk.

Finally, to be able fully overwrite files from the start, you must also implement the `edfs_truncate` operation. This function must set the file size to the given offset. If this offset is smaller than the current file size, redundant blocks must be released (a common operation is a truncation to zero size, which means all allocated blocks are released). The given offset can also be larger than the current file size, in which case new blocks must be allocated.

5 EdFS Specification

EdFS is based on the general concept of an inode-based file system. It is comparable to, for instance, *ext2*, but has been greatly simplified. Many details will not be considered such as permissions and timestamps, and the maximum file size that will be supported is limited. We do support file systems of different sizes and using different block sizes.

The following is an overview of the file system structure:



5.1 Super block

As can be seen in this overview, the super block is always located at an offset of 512 bytes. The super block has the following structure:

```
typedef uint16_t edfs_block_t;
typedef uint32_t edfs_inumber_t;

typedef struct
{
  uint64_t magic;
  uint16_t version;

  uint16_t block_size;
  edfs_block_t n_blocks;

  uint32_t bitmap_start; /* offset from start of device; in bytes */
  uint32_t bitmap_size;  /* in bytes */

  uint32_t inode_table_start; /* offset from start of device; in bytes */
  uint32_t inode_table_size;  /* in bytes */
  uint32_t inode_table_n_inodes;

  /* Inode hosting the root directory of the file system. */
  edfs_inumber_t root_inumber;
} __attribute__((__packed__)) edfs_super_block_t;
```

The magic number must be set to the value `0x00133700f00d0037`. The version field is ignored for now. The other fields indicate the block size used in the file system (for now 512, 1024, 2048, 4096 and 8192 bytes are supported), the total number of blocks that the file system contains (this includes the blocks on which boot block, super block, bitmap and inode table are stored), start and size of the bitmap (in bytes), start and size of the inode table (in bytes). Note that the sizes of the bitmap and inode table are always rounded up to the nearest block boundary. Therefore, you must use the `n_blocks` and `inode_table_n_inodes` fields to find out the number of valid entries contained in the bitmap and inode table respectively. Finally, the last item in the superblock is the number of the inode that describes the root directory.

Note that the starting point already reads the super block for you. You do not have to do this yourself. However, in order to implement the different functionalities of the file system you must be familiar with the fields in the super block.

5.2 Bitmap

In the bitmap the state of each block that is part of the file system is maintained. A block is either free (`0`) or allocated (`1`). The `bitmap_start` field of the super block indicates the byte offset within the file system where the bitmap starts. To read the status of a given block number, the corresponding bit number must be computed. Each bit in the bitmap represents one block. Since the bitmap data is stored as bytes, one byte represents 8 blocks: the least significant bit (bit 0) is the first block in the group of 8, the next (bit 1) is the second, and so on.

Note that the bitmap covers all blocks in the file system, also the blocks where the boot block, super block, bitmap and inode table are stored. So, the boot block is block zero and is always marked as occupied. This also holds for the blocks where the bitmap and inode table are stored. As a consequence, the offset of a block within the file system can simply be computed by multiplying the block number by the block size.

Furthermore, because block zero is always marked as occupied, the number zero is used as special identifier for unused/invalid blocks in inodes (`EDFS_BLOCK_INVALID`).

5.3 Inode table

All inodes are stored on disk consecutively starting at `inode_table_start`. The super block field `inode_table_n_inodes` indicates the number of inodes in the inode table. An inode on disk is stored according to the following structure:

```
typedef enum
{
  EDFS_INODE_TYPE_FREE = 0,
  EDFS_INODE_TYPE_FILE,
  EDFS_INODE_TYPE_DIRECTORY,

  EDFS_INODE_TYPE_INDIRECT = 1 << 7    /* Flag to indicate block pointers
                                        * are indirect blocks.
                                        */
} edfs_inode_type_t;

#define EDFS_BLOCK_INVALID 0
```

```
#define EDFS_INODE_N_BLOCKS 2   /* NB: Increasing this value will break
                                 * compatibility.
                                 */

/* Padded to be 16 bytes in size, with 5 reserved bytes available for
 * future expansion.
 */
typedef struct
{
  edfs_inode_type_t type : 8;
  uint8_t reserved[3];

  uint32_t size;  /* file size in bytes */

  edfs_block_t blocks[EDFS_INODE_N_BLOCKS];
  uint16_t reserved2[2];
} __attribute__((__packed__)) edfs_disk_inode_t;
```

For a valid (allocated) inode the `type` field *must* be set to either the file or directory type. The `blocks` array contains pointers to disk blocks in the form of disk block numbers. When unallocated, they must have the value `0` (`EDFS_BLOCK_INVALID`). The kind of block that is referred to depends on whether the bit `EDFS_INODE_TYPE_INDIRECT` is set in the `type` field. If *not* set, all entries of the `blocks` array point to data blocks of the file. Otherwise, if set, all entries point to indirect blocks, so in this case there can be up to *two* indirect blocks. An indirect block contains block numbers to data blocks. Within the indirect block these numbers are stored consecutively and therefore the contents of an indirect block can be seen as a small array of block numbers. Again, the special value `0` is used to indicate unallocated blocks. Note that a file may *not* contain unallocated holes.

The above implies that when the file size grows beyond `EDFS_INODE_N_BLOCKS` (in this case two) disk blocks, the inode needs to be converted from an inode storing direct block pointers to an inode storing indirect block pointers. To do so, a single indirect block is allocated and the block numbers `blocks[N]` for $0 \leq N < EDFS\_INODE\_N\_BLOCKS$ from the inode are stored as the first entries of this indirect block. Subsequently, the block number of the single indirect block is stored in `blocks[0]` and the other entries of `blocks` are set to `EDFS_BLOCK_INVALID`. Finally, the bit `EDFS_INODE_TYPE_INDIRECT` is set in the inode `type` field. When at a later point in time the file size grows larger than a single indirect block supports, additional indirect blocks (up to `EDFS_INODE_N_BLOCKS`) are to be allocated. Note that the reverse conversion, in case a file shrinks, is not required to be implemented.

To simplify implementation, directories *only* use direct blocks and on directory inodes the bit `EDFS_INODE_TYPE_INDIRECT` is never set. The inode with number `0` is left unused and must always be marked as free. Number `1` is the first valid inode. As a consequence, we can use `0` to signify invalid inodes in directory entries.

5.4 Directories

Within the file system directories are stored by writing directory entries to allocated disk blocks. Each directory entry has the following format:

```
#define EDFS_FILENAME_SIZE (64 - sizeof(edfs_inumber_t))
```

```
typedef struct
{
  edfs_inumber_t inumber;
  char filename[EDFS_FILENAME_SIZE];
} __attribute__((__packed__)) edfs_dir_entry_t;
```

For the current version of the file system the size of a directory entry is 64 bytes. Note that a directory inode may only allocate two direct blocks to store directory entries. For a block size of 512 bytes this means that a directory can store a maximum of 16 directory entries.

The `inumber` is an inode number which is used to read the corresponding inode from the 0-indexed inode table. The `filename` is restricted to 59 bytes (excluding null-terminator) and may only contain: A-Z, a-z, 0-9, spaces (" ") and dots (".."). Make sure to verify this when entering new files to the file system. A null-terminator must be stored, note that the structure allows for 60 bytes to be stored. If a directory entry is not in use, the `inumber` field must be set to zero (the invalid inode).

As an example, the root inode number indicated in the super block (often with a value of `1`) points to a valid inode with the directory type. This inode either has 0, 1 or 2 allocated blocks. Each of these allocated blocks consists of a sequence of consecutive directory entries. If the `inumber` of a directory entry is nonzero, it signifies a valid directory entry and the filename can be read from the `filename` field.