Operating Systems 2017, Assignment 3: File Systems

Deadline: Friday, May 12 before 23:59 hours.

1 Introduction

A disk can be accessed as an array of disk blocks, typically each block is 512 bytes in size. 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.

In this final assignment we will have a thorough look at a simple implementation of a file system, named WFS, and extend this implementation with full support for subdirectories and write support for files (the initial code that you will be provided with can only read files). For an excellent grade, we expect you to devise an advanced modification to the file system (see also below).

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[1]. 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. 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*.

2 Requirements

We expect the following to be achieved:

- Implement full support for subdirectories:
  - The user must be able to create/remove directories using *mkdir* and *rmdir*.
  - For each subdirectory two contiguous disk blocks are allocated to store its contents.
  - When creating new directories the filename must be verified for validity.
  - It must be possible to create/remove directories inside the root directory as well as inside subdirectories.
  - The implementation must be able to read and modify subdirectories on a file system image provided by us.
  - The structures on disk must be correctly updated. Changes must be persistent after re-mounting the file system.
  - *fsck.wfs* must always pass when run on an unmounted file system, also after the file system has been modified.

- Implement support for writing to existing files with non-zero length:
  - The file size must be correctly updated in the WFS file system on disk. Changes must be persistent after re-mounting the file system.
  - When necessary, new disk blocks must be allocated and be registered in the chain of blocks for the corresponding file. (Whether this is done in pairs or otherwise is up to you).
  - 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.

[1] `https://github.com/libfuse/libfuse`

  - *fsck.wfs* must always pass when run on an unmounted file system, also after the file system has been modified.

- Implement an advanced modification to the filesystem. For instance one of the following:

  - Advanced free-space management.
  - Storing directories using an interesting data structure.
  - Storing the allocated blocks of a file using a more advanced data structure compared to FAT that is currently used.

  Also include a very brief README describing the design. Good implementations are rewarded with a full point (to be able to score a 10), excellent implementations are also eligible for an additional bonus point.

- *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 2 persons. *Make sure that all files that you have modified contain your names and student IDs.* Please send us a "unified diff" that contains all modifications compared to the starting point. If applicable also include a README file in TXT format. To create the diff, have an unmodified copy of the starting point extracted as `wfsfuse.orig`, make sure all object files and binaries have been removed and run (make sure the filename contains your student IDs):

```
diff -upr wfsfuse.orig/ wfsfuse/ > lab3-sYYYYYYY-sXXXXXXX.diff
```

Check whether the *diff*-file contains *all* of the changes you have made and want to submit (open the *diff*-file in an editor for verification). Submit the *diff*-file by e-mail to *os2017 (at) handin (dot) liacs (dot) nl* and make sure the subject of the e-mail **equals** "OS2017 Assignment 3".

Deadline: We expect your submissions **before** Friday, May 12 before 23:59. No exceptions; deliveries after the deadline *will not be graded!*. Send e-mail attachments, Google Drive or DropBox links *are not accepted.*

We expect comments in the source code if the code is not immediately obvious, which often means you had to make a deliberate decision or trade-off. Do not add comments on things that are obvious. Note that we may always invite teams to elaborate on their submission in an interview in case parts of the source code need further explanation.

The maximum grade that can be obtained for this assignment is 10. The points are distributed as follows: Code Layout & Quality (1.0 / 10), Reading Subdirectories (1.0 / 10), Creating/Removing Subdirectories (3.0 / 10), Writing files (4.0 / 10), Advanced modification (1.0 / 10). Excellent implementations of an advanced modification are eligible for an additional bonus point.

4 FUSE

4.1 Using FUSE

Within the *os2017* environment we have prepared an installation of FUSE header files and testing utilities. Therefore, don't forget to enter the environment before working on the assignment:

```
source /vol/share/groups/liacs/scratch/os2017/os2017.bashrc
```

The starting point can be compiled using the supplied makefile. An initialized file system image can be obtained from the course website. 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:

```
./wfsfuse wfsimage.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 *wfsfuse*:

```
./wfsfuse -f -s wfsimage.img /tmp/testmyusername
```

This way, you can also run *wfsfuse* from within a debugger to debug your code.

4.2 FUSE on your own computer

We have tested the assignment on the University computers (Ubuntu 12.04), where FUSE version 2.8.6 is used. We expect no problems when a newer version of FUSE is used, but we have not tested this. There is also an implementation of FUSE for macOS, however due time constraints we could not test nor support this. If you have tested FUSE on another system, please let us know of your success stories!

4.3 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 `wfsfuse.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 `wfsfuse.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.

Several functions to help you implement the file system code are already provided, such as `wfs_find_entry`, `wfs_get_parent_entry` and `wfs_file_entry_operation`. Make sure to study these functions and understand how they can be used. These functions will also be mentioned in the text below.

4.4 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/`

5 The WFS File System

The WFS file system has been inspired by FAT, but has been greatly simplified. The file system has a fixed size, fixed amount of entries per directory, no support for user/group settings nor file permissions, and does not use a superblock.

The file system spans a fixed length of 8425488 bytes. At the start of this area are 16-bytes of magic numbers that are used for file system identification. This is followed by an area which contains the file entries of the root directory. There are 64 entries in total. Each file entry has the following format:

```
typedef struct
{
  char filename[58];
  uint16_t start_block;

  /* Given that the maximum size of a file is about 8MiB, we use the top
```

```
   * 4 bits of the size field for flags.
   */
  uint32_t size;
} __attribute__((__packed__)) wfs_file_entry_t;
```

The size of each entry is 64 bytes. Storing 64 of such entries requires 4096 bytes, or 8 512-byte disk blocks.

The 4 high bits of the `size` field are reserved for special information. Bit 32 is set when the entry describes a directory (instead of a file). The `start_block` field points out the first disk block (in the data section) that contains the file's content. Subsequent blocks can be found in the block table.

A filename is restricted to 58 bytes and may only contain: A-Z, a-z, 0-9 and dots ("."). Make sure to verify this when entering new files in the file system.

We have limited the file system to support a total of 16384 disk blocks. This means that at most 8 megabytes of information can be stored in a WFS file system. In order to know in which blocks the contents of a file are stored, we make use of the `start_block` field and the block table. The block table is indexed by `block - 1`, so `block_table[block - 1]` gives you the block that follows after `block`. There are two special block codes: `0x0` means that the block is free and can be allocated, `0xfffe` means that no block will follow the current block. So, to read a full file, you walk through the block table starting at the start block until the end of file code `0xfffe` is detected. A similar scheme is used in the FAT file system.

Given that we support at most 16384 blocks, the size of the block table is 2 bytes (`sizeof(uint16_t)`) times 16384, which is 32768 bytes. See also the `WFS_BLOCK_TABLE_SIZE` define in *wfs.h*. The table starts after the table with file entries for the root directory, see also the define `WFS_BLOCK_TABLE_START`.

After the block table follows the data area. The data area is large enough to hold 16384 blocks of 512 bytes. Remember that block 0 has a special meaning and is not stored, so to compute the offset of a block, you use $\texttt{WFS\_DATA\_START} + (block - 1) \times \texttt{WFS\_BLOCK\_SIZE}$ or specify $block - 1$ as argument to the function `wfs_get_block_offset()`.

Schematically, the file system has the following layout:

Magic numbers Size: 16 bytes
Root directory entries Size: 4096 bytes
Block table Size: 32768 bytes
Data area Size: 8388608 bytes

6 Subdirectory Support

We have seen that the entries for the root directory are stored at a designated location in the file system. By setting a certain bit in the size field, an entry can be made to indicate a directory instead of a file. An entry that indicates a directory is essentially a subdirectory and what is missing here is a place to store the file entries of that subdirectory. For subdirectories, the file entries should be stored in specially allocated disk blocks. When a new subdirectory is created, two *contiguous* disk blocks must be allocated. So, the number of file entries that can be stored in a subdirectory is 16. The block number of the first disk block is stored in the `start_block` field in the subdirectory's file entry. Make sure to terminate the chain for future compatibility.

6.1 Reading Subdirectories

In order to implement support for reading existing subdirectories, you need to modify the function `wfs_file_entry_operation`. You need to determine whether the given parent entry is the root directory of the file system, or a subdirectory. This is important, because you need to know how many directory entries the directory has and where these entries are located. The convention is that when the empty entry (note `wfs_file_entry_is_empty`) is passed as parent entry, the root directory is meant. Note that for the root directory no directory entry is stored in the file system. Additionally, `wfs_file_entry_operation` needs to be modified to read the correct number of file entries from the correct location.

6.2 Creating and Removing Subdirectories

To add the ability to create and remove subdirectories, the functions `wfs_mkdir` and `wfs_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 the WFS file system and whether the parent directory exists and is a directory. Also do not forget to use the various utility functions that are already provided: `wfs_get_parent_entry` and `wfs_get_basename`.

Implementing the *rmdir* function is easier. A single path is given as argument, which is the directory to remove. Look for the corresponding file entry in the table of directory entries of the path's parent. Verify that the directory to remove is indeed empty. Remove the entry from its parent and release all other resources that were allocated for this directory.

For both functions it will be helpful to extend the function `wfs_file_entry_operation` with new operations to add and remove file entries from a directory.

7 File Write Support

To implement write support for files, you will have to modify the function `wfs_write`. Before you start writing the code, we recommend that you study the code of `wfs_read` first. The following arguments are provided: `path` which is the path to perform the write operation to, `buf` which is the buffer to write to the file, `size` is the number of bytes to write, `offset` specifies the position in the file where the bytes should be written and finally `fi` provides some information about the file (which does not need to be used).

In summary, the purpose of the function is to write `size` bytes from the buffer pointed to by `buf` to the given `path` starting at the offset indicated by `offset`. When implementing this, it is important to be aware of the following:

1. `size` can be any length and does not have to be aligned on disk block boundaries.
2. You have to allocate new disk blocks using the block table when this is necessary.
3. Make sure to set the block table entry of the last block to `WFS_BLOCK_EOF` to indicate the end of a chain.
4. Your implementation of `wfs_write` must be able to cope with non-block-aligned writes.
5. Your implementation of `wfs_write` must be able to cope with seeks that are interleaved with write actions. What happens if a seek is performed past the end of the file followed by a write?
6. You only have to support writes to existing files with non-zero length. So, this guarantees that at least one block is already allocated to the file. Make sure to check this and bail out with an error if this is not the case! As a consequence of this, you do not have to write special code to deal with writes to new files that previously did not exist.
7. You do *not* have to support file truncation. If you write 20 bytes to the beginning of a 200-byte file, you can leave the remaining 180 bytes unmodified and do not have to throw these away. In other words, you only add support for growing of files and not for shrinking of files.

If you have modified the file size, you need to update the corresponding file entry on disk. To do so, write a function `wfs_file_entry_update` and make use of `wfs_get_parent_entry` and `wfs_file_entry_operation`. It is fine if you keep it simple and simply update the size in the entry for every write action (of course this is something that can be optimized!).

8 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.

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

9 Testing

We have provided several utilities to test your code. We strongly recommended to start the assignment by implementing subdirectory support. To test support for reading subdirectories, use the initialized file system image provided on the website. 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 WFS image is also available from the website.

To test the support for creating and removing subdirectories, you can make use of 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 WFS file system as well as creating subdirectories in subdirectories (nesting).

For testing file write support, you can use the *overwrite* utility, which is present in `PATH` when in the *os2017* environment. 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

Important: 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 can use the *fsck.wfs* utility (also available in the *os2017* environment). 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 and report that the file system is "clean". Though, remember that the file system check might not catch every possible error! Finally, note that in case you implement one of the specified advanced features, you will modify the format of the file system and because of this the provided file system checker need no longer apply!