Operating Systems 2014 Assignment 4: File Systems

Deadline: Sunday, May 25 before 23:59 hours.

1 Introduction

A disk can be accessed as an array of disk blocks, often each block is 512 bytes in length. 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 blocks will we write a given file? 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 and extend this implementation. You may have noticed that a file system named WFS has been used throughout this lab. You will extend this file system with write support for files (as the file system currently is read only) and full support for subdirectories.

2 Requirements

You may work in teams of at most 2 persons. You have to modify existing operating system source code written in C and therefore you will be writing C code. Your submissions must adhere to the following requirements:

- Submit the source code of the operating system with your functioning extended WFS implementation.
- Your implementation should contain full support for subdirectories:
  - The user must be able to create/remove directories with the provided *mkdir* and *rmdir* utilities.
  - 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 ramdisk image provided by us.
  - Both the structures on disk as well as the VFS structures must be updated. Changes must be persistent after unmounting the file system and rebooting the system.
  - *fsck.wfs* must always pass when run on an unmounted file system, also after the file system has been modified.
- Your implementation must be able to write to existing files. This includes:

  - The file size must be updated in the WFS file system on disk as well as in the VFS data structures. Changes must be persistent after unmounting the file system and rebooting the system.
  - We have provided a utility called *overwr* which overwrites a given file. See also the section on Testing below.
  - After using the *overwr* utility, the *cat* utility must be able to show the correct contents of the file (without modification to the *cat* utility) before unmounting and after unmounting and restarting.
  - The *overwr* utility must work on files in both the root directory as well as subdirectories.
  - *fsck.wfs* must always pass when run on an unmounted file system, also after the file system has been modified.

- *Important!* Make sure to use extensive error handling in your code so that your code detects various kinds of failures. For example, when writing to a file system, return appropriate error codes when the file system is full, a given filename is invalid (too long or contains invalid characters), when you are asked to create a directory in a node which is not a directory, etc.

When grading the submissions, we will look at whether your source code fulfills these requirements and the source code looks adequate: good structure, consistent indentation and comments where these are required. Comments are usually required if the code is not immediately obvious, which often means you had to make a deliberate decision or trade-off. Document these decisions, trade-offs and why in the source code. Note that we may always invite teams to elaborate on their submission in an interview in case parts of the source code are not clear.

Make sure that the files you have modified contain your names and student IDs. Remove any object files and binaries by removing the build directory. Create a gzipped tar file of the directory:

```
tar -czvf assignment4.tar.gz assignment4
```

Mail your tar files to *krietvel (at) liacs (dot) nl* and make sure the subject of the e-mail contains *OS Assignment 4*. Include your names and student IDs in the e-mail.

Deadline: We expect your submissions by Sunday, May 25 before 23:59.

3 Kernel

We will use the same kernel as with the second and third assignments, however, you will be provided with a new starting point and the SD card setup is different for this assignment. Make sure to download the pre-initialized SD card image from the course website.

Recall that in the previous assignments, we used an SD card image that contained a single FAT16 file system that held the kernel image, ramdisk image and boot script. The boot loader would load the ramdisk image (which was formatted as WFS) into memory and the kernel would use this as its root file system.

In this assignment an SD card image with two partitions will be used. The first is a FAT16 partition that will contain the kernel image, boot script and utilities such as *sh* and *ls*. This partition will *no longer* contain a ramdisk image. Instead, the FAT16 partition is used as root file system directory. The second partition on the SD card image is formatted with WFS. When the system has booted, you can mount the WFS partition under the */mnt* path using the *mountfs* command (no arguments required). *unmountfs* unmounts the file system again.

Because of the different SD card layout, you also need to use different commands to access the partitions on the image. To copy a new kernel image to the FAT16 partition, use:

```
mcopy -i sdcard.img@@1048576 kernel.bin ::
```

Other *mtools* commands work similarly. Note that relatively recent *mtools* is required to be able to use the @@ notation. Also *wfstool* has been adapted and now accepts the same notation as *mtools*. So, for example (note the different file offset!):

```
kernel/utils/wfstool/wfstool sdcard.img@@32505856 list
kernel/utils/wfstool/wfstool sdcard.img@@32505856 put file1.txt file1.txt
```

4 Limitations

As the file system code is still in its infancy, you should be aware of a couple of limitations. Concurrent access to the file system (by different processes) is not possible. This will be corrected in the future by introducing proper locking in the file system code. Also we do not support setting owning users, groups and permissions.

5 Virtual File System

Like many other kernels, this kernel makes use of a Virtual File System (VFS). The Virtual File System is an in-memory representation of the system's file system. This representation has a data structure in the form of a tree.

A VFS is used for two main reasons. Firstly, performance, by storing file nodes in a VFS tree we do not have to read file entries from disk each time we want to list a directory or get the size of a file. Secondly, because of support for mount points. You can mount a file system at a given directory in another file system. For example, in our kernel a "devfs" file system is mounted under `/dev`. Thirdly for flexibility. Using VFS, we have an overview of the entire file system of the system (which comprises multiple file systems) as well as a generic interface for accessing files and directories. The VFS will take care to call the functions of the correct file system implementation in order to carry out the desired operation on the file or directory. The interface makes it easy to implement file systems which is essentially done by implementing the functions in the VFS file ops and file sys structures.

The VFS consists of a number of fundamental types: `vfs_vnode_t` that represents any file in the VFS, including normal files, mount points, directories and devices. `vfs_file_t` that represents an open file, that is, it has a VFS node and an offset where the current read / write pointer is located in the file. Further, each file system implements a number of file operations. These are stored in the type `vfs_fileops_t`. This structure contains what would be called virtual functions in some programming languages. Essentially, when calling the `vfs_open` function, the call will be dispatched to the open function, whose address is stored in the file ops structure for the given vnode. The next important type is `vfs_filesys_t` that implements the basic interface for mounting and unmounting the file system; it also contains the function that returns the root node of a file system.

6 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 and file permissions and does not make use of 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.

We have limited the file system to support 16384 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 $\texttt{WFS\_DATA\_START} + (block - 1) \times \texttt{WFS\_BLOCK\_SIZE}$.

Schematically, the file system has the following layout:

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

7 Subdirectory Support

We have seen that the entries for the root directory are stored at a special place 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 a specially allocated disk block. We will keep it easy and limit the amount of file entries in a subdirectory to 8. Given that 8 times 64 makes 512, this fits exactly in a single disk block. The block number is stored in the `start_block` field in the subdirectory's file entry. Make sure to terminate the chain for future compatibility.

7.1 Reading Subdirectories

In order to implement support for reading existing subdirectories, you need to modify the function `wfs_directory_readdir`. You need to determine whether the given node 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. You can find out the directory type by looking at the node's parent, if this node is of type `VFS_MOUNT` you know this node is the root directory, otherwise it is a subdirectory. Finally, you modify the function to read the correct number of file entries from the correct location.

7.2 Creating and Removing Subdirectories

To add the ability to create and remove subdirectories, the functions `wfs_directory_mkdir` and `wfs_directory_rmdir` should be implemented and registered in the `wfs_directory_fileops` structure. Look for the exact prototypes of these two functions in `vfs.h`.

The *mkdir* function has a parent node (which must be a directory, do not forget to verify this) and a name string as argument. A subdirectory with the name string as name should be created in the file system. Remember to update the metadata on the disk itself as well as the VFS data structures. For the latter, the function `vfs_dirp_addnode` will come in handy.

Implementing the *rmdir* function is easier. A single node is given as argument, which is the directory to remove. Look for this node in the table of directory entries in the node's parent and remove the node. Again update the metadata on disk as well as the VFS data structures and for *rmdir* the function `vfs_dirp_removenode` will be of use when updating the VFS data structures.

8 File Write Support

To implement write support for files, you will have to modify the function `wfs_file_write` in *wfs.c*. Before you start writing the code, we recommend that you study the code of `wfs_file_read` first. As argument you get a pointer to a `vfs_file_t` structure, which has a `vnode` field pointing out the node's type and `fs_data` and an `off` field that indicates the current read/write offset into the file.

The purpose of the function is to write `len` bytes from the buffer pointed to by `buff` to the given `file` starting at the offset indicated by `file->off`. When implementing this, it is important to be aware of the following:

1. `len` 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_file_write` must be able to cope with non-block-aligned writes.
5. Your implementation of `wfs_file_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?

If you have modified the start block or the file size, update the fields in the VFS node (`file->vnode`) and FS data (`file->vnode->fs_data`) and write a function `wfs_update_node(vfs_vnode_t *vnode)` which will update the given `vnode` in the file system metadata on disk. This update function should be called when the file is closed (`wfs_file_close`).

9 Testing

We have provided several utilities to test your code. For testing file write support, you can use the *overwr* utility. You need to specify an *existing* file name as an argument (for now it does *not* accept absolute paths). The utility will overwrite the specified file with a particular pattern. By default this pattern is 3550 bytes in length.

Note that the initialized SD card 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 *overwr* (*overwr* compares the filename), a special action will be performed that is defined for that specific file. Ensure to test your code by using *overwr* on each of the five test files. After overwriting the file, inspect the new file size with *ls* and its contents with *cat*. Also unmount the file system (and optionally reboot) and mount it again and verify the made changes are persistent.

To test support for reading subdirectories, use the initialized SD card image. You should be able to inspect a hierarchy of subdirectories using *ls* and *cd*.

Finally, 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 as an argument. Like the *overwr* utility they do not accept absolute paths. You should test creating subdirectories in the root directory, but also creating subdirectories in subdirectories (nesting).

To verify the file system is not corrupted after making changes to it, you can use the *fsck.WFS* utility. This is a file system checker that is provided with the initialized SD card image for this assignment. This utility ensures that the data on disk is correct and not corrupted. It should report no errors and that the file system is "clean". Though, remember that the file system check might not catch every possible error!