Assignment 3: Filesystem

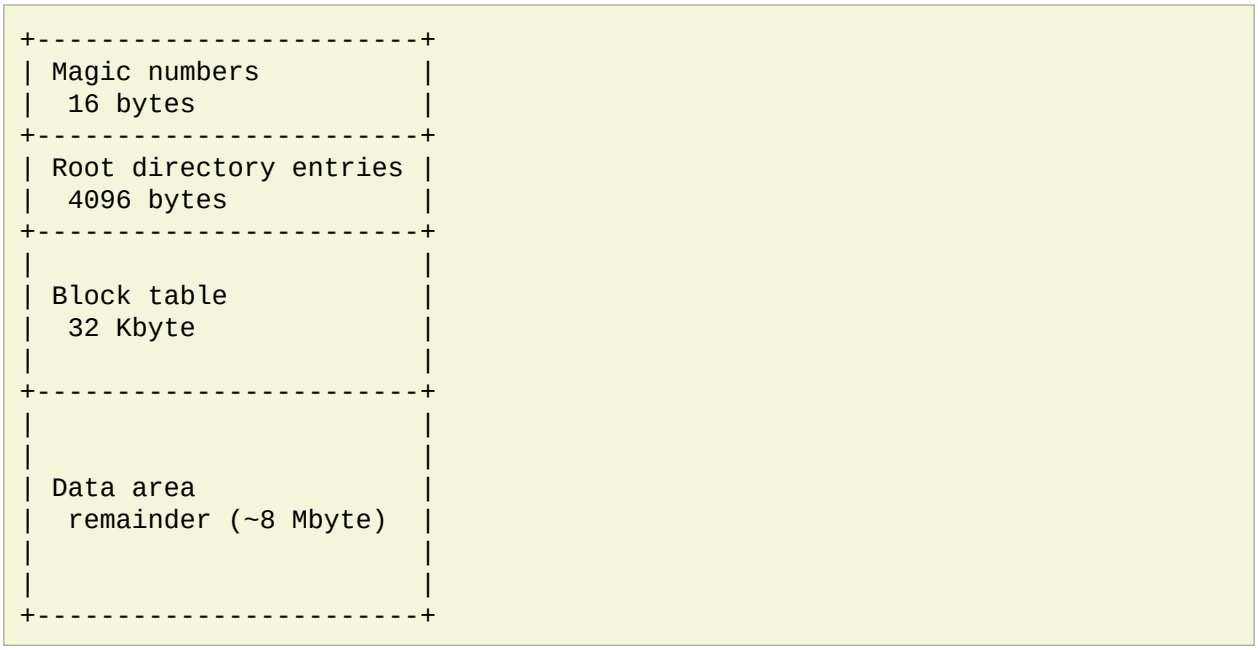
In this assignment you will implement your own driver for a new filesystem: SFS. This simple filesystem is loosely based on the FAT filesystem, and supports (sub)directories and files of arbitrary size. It does have some severe limitations, such as a maximum filesystem size of 8 MB, only a limited number of items per directory, and no metadata such as file permissions.

Your task is to write a driver for this filesystem, so that files and directories can be navigated, read, created, removed and modified. Writing a filesystem directly on a disk partition would be too complicated for this assignment, and therefore your SFS filesystem is contained in an image. This image resides on your normal (host) filesystem, and the driver you write will read and write to this image as if it is a disk.

Normally filesystem drivers are part of the operating system. However, for this assignment we will implement the driver in userspace, through a framework called **FUSE**. With FUSE, drivers are implemented as a normal (userspace) binary, and are called by the FUSE kernel driver when a user interacts with your files. By using FUSE, you can actually use your driver to *mount* the SFS filesystem under a directory, then allowing you to navigate into it using your favorite file browser, or `cd` into it with a terminal and `cat` your files.

Layout of SFS

The diagram below shows a conceptual overview of an SFS partition/image (not to scale). Note that all datatypes and information about the layout here are also provided in `sfs.h` in the framework, for your code to use.



Each SFS partition starts with 16 bytes of **magic numbers**. These identify the partition as SFS, and if these numbers are not found by your driver or any of the SFS tools (described later), they abort to prevent corrupting data.

After the magic numbers follows the contents of the **root directory** of the SFS partition. The entries in the root directory point either to files or subdirectories. The format of a directory entry is described later in this section.

Following the root directory is the **block table**. Each file and subdirectory keeps its data in **disk blocks** of 512 bytes. Files consisting of multiple blocks do not necessarily have consecutive data, and this block table describes, for each block on disk, what the *next* block is. For the last block in a chain (i.e., the last block in a file) a special marker is used: `SFS_BLOCKIDX_END` (`0xfffe`). Additionally, a value of

`SFS_BLOCKIDX_EMPTY` (`0xffff`) in this table indicates a block is unused, and can thus be allocated when creating a file or directory. The block table has space for `0x4000` entries.

The data area spans the rest of the disk, and consists of `0x4000` blocks of 512 byte each, resulting in 8 MB of data.

Directory entry format

Each directory consists of a fixed number of entries, each representing a file or subdirectory (or unused space). These are described by the following layout (see `sfs.h`):

```
typedef uint16_t blockidx_t;

struct sfs_entry {
   char filename[58];
   blockidx_t first_block;
   uint32_t size;
} __attribute__((__packed__));
```

The `filename` of an entry is the name of the file or subdirectory. It is a null-terminated string, and thus can contain at most 57 readable characters (excluding special characters such as '/').

The `first_block` field points to the first block index used by the file or subdirectory. For example, a value of 3 in this field means the first part of the data is contained in the 4th disk block (because we start at 0), i.e., at offset `512 * 3` from the start of the data area. To find subsequent blocks of this entry, consult the block table. For empty files, the `first_block` field contains the end-of-file marker.

Finally the `size` field describes the size of a file in bytes, **and** any special flags (in the upper 4 bits). In particular, if the upper bit of this field is set, this entry refers to a directory instead of a file. In your code, you can use the `SFS_SIZEMASK` and `SFS_DIRECTORY` macros for this. When an entry describes a subdirectory (i.e., has its upper bit set) then the size-mark (i.e., the lower 28 bits) must be all zero.

So for **subdirectories** there an exists a directory entry in its parent directory with the upper bit of the size field set. Each subdirectory consists of two *consecutive* blocks on disk, where the `first_block` field points to the first one. The block table entries for these two blocks still describe the blocks the same as files (i.e., the first block points to the 2nd block, and the 2nd block points to the end-of-file marker).

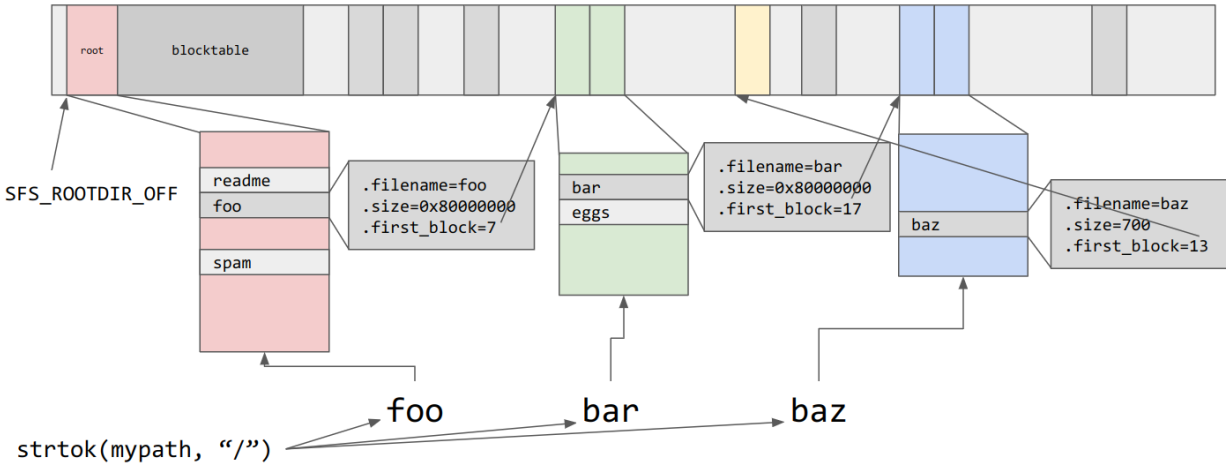
The total size of this directory entry struct is 64 bytes. Since the root directory is 4096 bytes, it contains 64 (possible) entries. Subdirectories are smaller (two 512 block, so 1024 bytes total) and only contain 16 entries.

Unused entries in a directory should set filename to all null characters, size to 0 and first_block to `SFS_BLOCKIDX_EMPTY`.

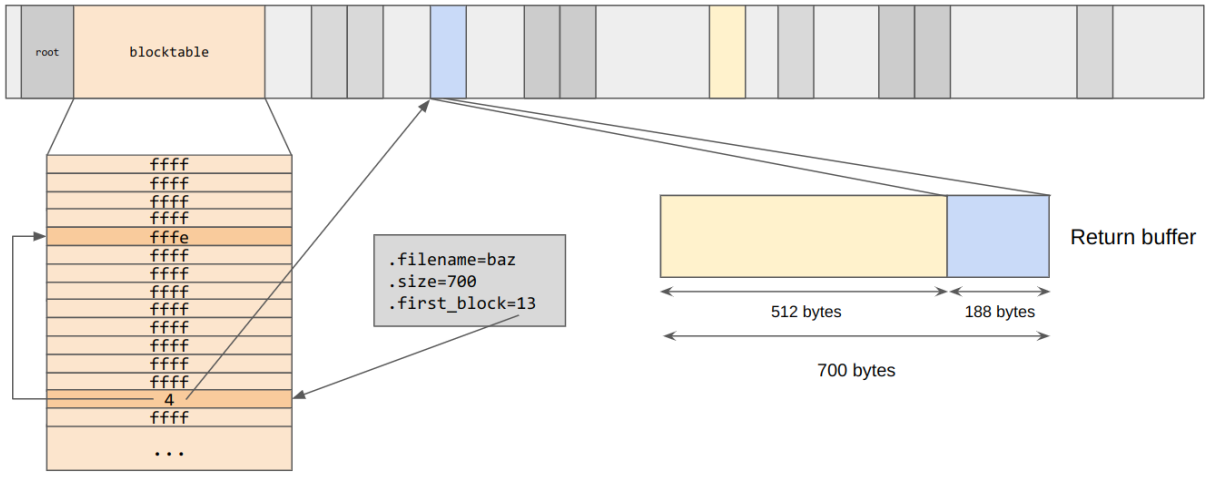
An example

Consier the path `/foo/bar/baz`. How do we read the file?

1. Use `strtok()` to split up the path (please refer to the man page). At the first iteration `strtok(mypath, "/"")` returns the token `foo`.



2. Iterate over the root directory entries. This is an array of `sfs_entry` that starts at offset `SFS_ROOTDIR_OFF`. Find the entry whose `filename` matches with `foo`.
    - Note that the `size` has the upper bit set to one to indicate that it is a directory. The field `fist_block` specifies the index of the first data block.
    - Directories are made of two data blocks: we should read from the data blocks at index 7 and 8.
    - Given `blockidx` - the index of the data block - you can find the offset by computing `SFS_DATA_OFF + blockidx * SFS_BLOCK_SIZE`, where `SFS_DATA_OFF` is the offset of the **Data area** in the disk image.
3. Read the data blocks at index 7 and 8; these data blocks contain the entries for `foo`, namely its subdirectories and files.
    - Calling `strtok()` again returns `bar`
    - Find the entry for `bar`; the first block is at index 17.
4. Read the data blocks at index 17 and 18. Iterate over the entries and find the one for `baz`.
5. The `size` field for the entry of `baz` has the upper bits unset: this is a file that contains 700 bytes. Each *disk block* is 512 bytes (i.e., `SFS_BLOCK_SIZE`) therefore we need to read 2 blocks. The first one is located at index 13.
6. To find the second block, look at the *blocktable*. The blocktable is an array of `blockidx_t`. The `blockidx_t` at index 13 contains the index of the second block for `bar` (i.e., 4).



7. Read the remaining 188 bytes (700-512) from the data block at index 4. This is the last one for the file `bar`. Indeed the `blockidx_t` at index 4 contains the special value `fff3`.

SFS tools

For this assignment we provide two tools to create and inspect SFS images. These tools are invaluable when developing and debugging your driver, so it is good to familiarize yourself with them.

Creating disk images with mkfs

The `mkfs.sfs` binary produces a valid SFS image (that you can mount) with any contents you specify. For all supported options, see `.mkfs.sfs --help`.

To produce an image with this README, and empty directory foo, and an empty file bar/baz, you can run the following command:

```
$ ./mkfs.sfs test.img /README:README.rst /foo/ /bar/baz
Creating fresh SFS filesystem
Creating file '/README' from host file 'README.rst'
Creating empty file '/bar/baz'
```

Some basic rules on the syntax of the arguments:

- First is always the name of the entry inside the image, always starting with a slash ('/').
- Any entry ending with a slash ('/') describes a directory.
- For files, an optional argument can be specified using a colon (':'). Without this optional argument, the file will be empty. With this argument the file inside the SFS image will be created with the contains of the filename on the host filesystem. In this example, inside the SFS image we get a /README file with the contents of the host file README.rst.

Inspecting disk images with fsck

The `fsck.sfs` binary performs file system checks on SFS images, and can additionally print its contents. See `./fsck.sfs --help` for a list of all supported options.

By default the tool only performs (silent) checks, and will not produce output unless an error is found. With the `-l` flag, it will print all files and directories in the image, e.g.:

```
$ ./fsck.sfs test.img
$ ./fsck.sfs -l test.img
00005373 0000  /README
80000000 002a  /foo/
80000000 002c  /bar/
00000000 fffe  /bar/baz
```

The first field printed is the size of the entry (in hex). Notice the uppermost bit is set for directories). The second field is the first block of the entry (in hex). Finally, the full path of the entry is printed.

For inspecting images in more detail, the `-d` flag will print the md5sum of each file, the `-c` flag prints the full contents of each file, and the `-b` flag dumps the block indices of the blocklist.

The `-v` flag enables (very) verbose debug output. If fsck is reporting errors and you want to inspect the situation in more detail, this can be useful.

Using FUSE

FUSE allows for userspace binaries to implement drivers, that are indirectly used by the kernel. This allows you to mount a partition, image (or other source, like a network share) onto your filesystem. The file `sfs.c` produces, when built, the `sfs` binary which will call into FUSE. This means that you mount an SFS image by running your `sfs` binary.

Installing FUSE and building

For this assignment you can work natively on Linux or WSL2, but we also offer a Docker container that should work correctly with FUSE when invoked with higher privileges. When using `make docker-check` the `--privileged` flag is automatically passed to docker. For this to work correctly you may need to install FUSE first on the host (e.g., `sudo apt install fuse libfuse-dev`).

If you are using Docker (e.g., because you're on macOS), you may also also want to use the docker container interactively, for example to play with FUSE like described below. For this we recommend the following command:

```
$ docker run --privileged -i -t --rm -v "`pwd`:/code" -w /code \
      vusec/vu-os-fs-check /bin/bash
```

This will launch a docker running bash, with your current host directory mounted at `/code`. On Linux you probably want to add the `-u `id -u`:`id -g`` flag, so files on your host are not suddenly owned by `root`. **Important: any changes *outside* the /code directy are lost when you exit the container.**

After installing the dependencies (or dropping into the docker container), you can (re)build your code by simply running:

```
$ make
```

Mounting your image through FUSE

After building the `sfs` binary you can mount an image simply using:

```
$ mkdir mnt
$ ./sfs -v -i test.img mnt
 # getattr /.Trash
 # getattr /.Trash-1000
```

The `-v` flag enables some debug logging (as can be seen in `sfs.c`), and in this case shows the **callbacks** that FUSE is calling into your application when the kernel asks for this. This is how FUSE works: every action a user does on files goes through the kernel via system calls (e.g., `read`, `write`, `mkdir`, `readdir`). Linux forwards these to FUSE, which in turn forwards them to your program.

One of the most fundamental calls within FUSE is the `getattr` callback. This asks your driver for information on a file or directory, including whether it exists and, if so, its properties (e.g., is it a file or directory, what is its size, etc). In the above example we saw two calls to this to this function, which is Gnome detecting a new partition was mounted, and checking if there exists a "trash bin" on it. Our driver can say no by returning the error code `-ENOENT`.

Let's try another example, by opening another terminal on the side:

```
$ ./sfs -v -i test.img mnt
 # getattr /.Trash
 # getattr /.Trash-1000
                              $ ls mnt/
```

```
# getattr /
# readdir /
                              ls: reading directory 'mnt': Function not implemented
                              $ cat mnt/somefile
# getattr /somefile
                              cat: mnt/somefile: No such file or directory
                              $
```

So we can't do much yet, but it demonstrates that simple programs like `ls` and `cat` are simply asking our driver about the filesystem. For the `ls` example, it first checks if '/' exists in our image. The skeleton implementation in this framework reports that it does, and thus `ls` goes on to read its directory contents. This function is *not* implemented (it returns `-ENOSYS`), and this is what `ls` prints. When we try to read some file with `cat` we can see that `cat` is asking if the file exists. Our skeleton `getattr` function returns `-ENOENT` and thus `cat` thinks the file does not exist.

Try playing around with different programs to see what they do, especially after implementing a basic version of `getattr`.

Interacting with the disk from your code

Your driver has to interact with the underlying storage device that contains the SFS partition. For ease-of-use we use an image instead of a real disk partition. To interact with the (virtual) storage device, the framework contains an interface that can be found in `diskio.h`. In particular:

```
void disk_read(void *buf, size_t size, off_t offset);
void disk_write(const void *buf, size_t size, off_t offset);
```

The `disk_read` function reads bytes from the disk into the provided buffer `buf`. The function will read `size` bytes, and it will start reading from the disk at offset of `offset` bytes.

Similarly, the `disk_write` function writes `size` bytes of the provided `buf` onto the disk at `offset`.

You can find offsets for particular SFS areas in `sfs.h` (e.g., `SFS_BLOCKTBL_OFF`). To access the 4th block of the data area (blockidx 4), you would read at offset `SFS_DATA_OFF + 4 * SFS_BLOCK_SIZE`.

Important: you *must* use these functions to read and write to/from the underlying storage device (disk/image). Additionally, you should do this for **every operation**. You are *not* allowed to read the entire contents of the disk into memory, operate in memory, and write the whole thing back.

For example, if we want to read file `/foo`, we would first issue a `disk_read` at `SFS_ROOTDIR_OFF` to read the contents of the root directory. In the resulting data we look for an entry with the name `foo`. To then read the contents of the file, we read the first 512 bytes with a `disk_read` call at the specified blockidx in the data area. Then we need to find the next blockidx of the file, we issue a `disk_read` into the blocktable, and we repeat calling `disk_read` to read data blocks and blocktable entries until we read the entire file.

Accesses to the disk with `disk_read` and `disk_write` do *not* have to be block-aligned. Normally on physical storage devices, a driver has to read a whole sector at a time in 512-byte aligned blocks. We do not have such a constraint for this assignment, and you *are* allowed to read, for example, just 2 bytes from the middle of the block table on disk.

The assignment and grading

This assignment is individual; you are not allowed to work in teams. Submissions should be made to the submission system before the deadline. Multiple submissions are encouraged to evaluate your submission on our system. Our system may differ from your local system (e.g., compiler version); points are only given for features that work on our system.

Your grade will be 1 if you did not submit your work on time, has an invalid format, or has errors during compilation.

If your submission is valid (on time, in correct format and compiles), your grade starts from 0, and the following tests determine your grade (in no particular order):

- +1.0pt if you made a valid submission that compiles.
- +0.5pt for implementing the `readdir` function that works on the root directory. **Required**
- +1.5pt for implementing functionality to read files in the root directory.
- +1.0pt for supporting subdirectories (for `readdir` and `read`).
- +1.0pt for implementing support for `mkdir`.
- +1.0pt for implementing support for `rmdir`.
- +1.0pt for implementing support for removing files through `unlink`.
- +1.0pt for implementing support creating (empty) files.
- +1.5pt for implementing support for `truncate` to shrink and grow files.
- +2.0pt for implementing support writing to files.
- -1.0pt if `gcc -Wall -Wextra` reports warnings when compiling your code.

If you do not implement an item marked with **Required** you cannot obtain any further points.

The grade will be capped at 10, so you do not need to implement all features to get a top grade.

To get an indication of the grade you might get, you can run the automated tests using the command `make check`.

Note: Your filesystem driver will be evaluated largely automatically. This means features only get a positive grade if they work perfectly, and there will be no half grade for "effort".

Notes and hints

- The header file `sfs.h` should contain all information about the layout of the SFS filesystem for your code to use. Make sure you understand all constants and types defined in this file.
- Make sure to properly detect error conditions (e.g., a filename that is too long, a directory that is full, removing a non-empty directory, etc) and return the appropriate error code.
- Remember that the `getattr` callback lies at the core of most FUSE operations, and you will have to properly implement it for other functions to work. For example, FUSE will not even bother calling `readdir`, `read` or `mkdir` if the appropriate (parent) entry is not correctly reported by `getattr`
- To test `getattr` separately from the terminal, you can use the `stat` command (e.g., `$ stat mnt/foo/bar`).
- To test the `offset` parameter of `read` manually from the terminal, you can use the `dd` command. E.g., `$ dd if=mnt/foo/bar bs=1 skip=123`, where the `skip` number is passed as offset to `sfs_read`. Similarly for `write` you can use `$ dd if=somefile of=mnt/foo/bar bs=1 seek=123 count=456`, where which will write `count` bytes from `somefile` into `mnt/foo/bar` at offset `seek`.
- When doing manual tests with **Docker**, refer to the setup document on how to open multiple terminals with the same Docker session.
- Remember that you should check for empty (non-existent) directory entries by looking at the filename field (e.g., `strlen(entry.filename) == 0`), **not** by looking at the size (which can be 0).

- For most functions you will need to start with finding the correct directory entry corresponding to the `path`. Especially later when you add support for subdirectories, it is advised to create a reusable function to do this. You can for example create a function like:

```
int get_entry(const char *path, struct sfs_entry *ret_entry,
              unsigned *ret_entry_off)
```

  This function would split up the `path` (using `strtok`), and recursively walk down the directories. The result is placed in `ret_entry`. Additionally, you may want to add an additional return value which describes where on the disk the returned entry was found, in case you need to modify it and write it back (e.g., for `rmdir` or `write`). For this purpose, in the example above the offset on the disk is returned via `ret_entry_off`, but there are multiple ways of doing this. Note that you do not have to use this function, or can add/change arguments however you want - this is just a hint on how to easily organize your code.

Troubleshooting

- Everything works fine when testing manually, but the tests all fail: The most common cause is the randomization of the image layout that the tests use. When creating images for tests, `check.py` passes the `-r` flag to `mkfs.sfs`, which cause randomization of which blocks to use for files and directories, and causes directory entries to use random slots (instead of starting at the first entry). To support this, you have to make sure you walk all directory entries when looking for a path, and you have to use the blocktbl to find the next block for each file. You can apply this randomization yourself by also passing the `-r` flag to `mkfs.sfs`.
- If your `sfs` binary crashes FUSE might not properly unmount your directory. In these cases, use the following command to unmount it: `fusermount -u <DIR>`
- "Transport endpoint not connected" errors: This happens when your driver (sfs binary) crashed. If you see this error during the automated tests, try running the ./sfs binary manually and reproduce what the tests were doing.
- Random "Input/output error" (even when you never return -EIO): In most cases this happens when you modify the `path` variable given by FUSE to most functions. This variable is marked as const, and should **not** be modified (e.g., using `strtok`). Make a copy (using `strdup`) before modifying it.
- "mounting over filesystem type 0x01021997 is forbidden" (on **WSL2**): The mountpoint (i.e., the parameter passed to ./sfs) should **not** be inside the `/mnt/c` part of the filesystem (the windows disks, which are mounted -secretly- over the network). Place your mountpoint somewhere in `/home` or `/tmp` instead.
- Slow tests on **WSL2** (`make check` should finish in about 10 seconds): Place all you files outside of the windows filesystem (`/mnt`) and instead in the local home directory (`/home`).
- "fuse: device not found, try 'modprobe fuse' first" (on **WSL**): You are using WSL1, not WSL2. On WSL2 FUSE should work out of the box.