Project 2: Simple and Safe File System (SSFS)

INFO0940-1: Operating Systems (Groups of 2)

Submission before Tuesday, 13th May 2025 (23:59:59 CET)

1 Introduction

In this assignment, you will implement a simple file system. Your implementation, in C, will run in user space and will interact with a provided virtual disk driver for all the needed operations with the disk.

2 Virtual disk

The disk is emulated by dividing a (disk image) file into fixed-size sectors. This emulation is the job of the provided virtual disk driver code so you do not need to worry about it.

Assume that the sector size is 1024 bytes.

The virtual disk driver API is composed of the following functions:

```
int vdisk_on(char *filename, DISK *diskp);
int vdisk_read(DISK *diskp, uint32_t sector, uint8_t *buffer);
int vdisk_write(DISK *diskp, uint32_t sector, uint8_t *buffer);
int vdisk_sync(DISK *diskp);
void vdisk_off(DISK *diskp);
```

- `int vdisk_on(char *filename, DISK *diskp)` must be called once before any further use of the virtual disk.
  - `filename` is a null-terminated string specifying the disk image file to be used.
  - `diskp` is a "return parameter", that is a pointer to caller allocated space, which then gets initialized, with data of type `DISK`, by `vdisk_on`. This pointer must then be passed to all the other functions of the virtual disk driver, as a way to identify the virtual disk.
  - Return value: 0 on success; `vdisk_EACCESS` if the user does not have appropriate permissions to manipulate the disk image file; `vdisk_ENOEXIST` if the disk image file does not exist; `vdisk_ENODISK` if the size of the disk image file is smaller than the size of a sector; -1 in the event of an unknown error[1].
- `int vdisk_read(DISK *diskp, uint32_t sector, uint8_t *buffer)` reads a sector from the virtual disk.
  - `diskp` identifies the virtual disk
  - `sector` is the sector number of the virtual disk sector to be read (sectors are numbered sequentially from 0)

[1] In this project, all error codes are negative integers.

  - `buffer` is a pointer to a buffer that will receive the content of the sector being read. It is the programmer's responsibility to ensure the buffer has enough capacity.
  - Return value: 0 on success; `vdisk_ESECTOR` if the sector could only be read partially (damaged sector); `vdisk_EEXCEED` if the requested sector is beyond the end of the virtual disk; `vdisk_ENODISK` if `vdisk_on` was either not called or failed, or `vdisk_off` was called, prior to this call.

- `int vdisk_write(DISK *diskp, uint32_t sector, uint8_t *buffer)` writes content to a sector on the virtual disk.
  - `diskp` identifies the virtual disk
  - `sector` is the sector number of the virtual disk sector to be written (sectors are numbered sequentially from 0)
  - `buffer` is a pointer to a buffer with the content of the sector to be written. A number of bytes equal to the sector size is written.
  - Return value: 0 on success; `vdisk_ESECTOR` if the sector could only be partially written (damaged sector); `vdisk_EEXCEED` if the requested sector is beyond the end of the virtual disk; `vdisk_ENODISK` if `vdisk_on` was either not called or failed, or `vdisk_off` was called, prior to this call.
- `int vdisk_sync(DISK *diskp)` transfers ("flushes") all modifications to the virtual disk image.
  - `diskp` identifies the virtual disk
  - Return value: 0 on success; `vdisk_ENODISK` if `vdisk_on` was either not called or failed, or `vdisk_off` was called, prior to this call.
- `void vdisk_off(DISK *diskp)` discards all modifications that have not yet been transferred to the virtual disk image. After this call, the virtual disk can no longer be used without a new call to `vdisk_on`.
  - `diskp` identifies the virtual disk

Note that none of these functions can change the size of the disk image file.

3 SSFS Specification

SSFS has a very simple structure depicted in Figure 1.

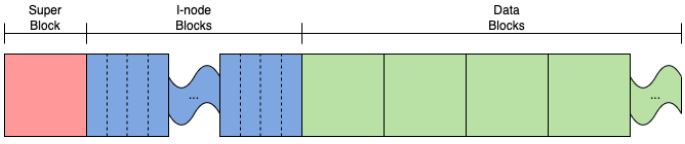

Figure 1: SSFS Structure.

Blocks are numbered sequentially, starting from 0 at the Super Block. In SSFS, the size of a block is equal to the size of a disk sector.

SSFS is said to be safe because everything is always zero, unless other values are strictly needed.

3.1 Super Block

The Super Block is depicted in Figure 2.

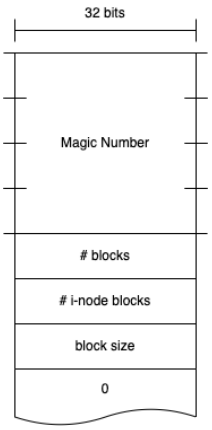

Figure 2: Super Block.

The magic number is a 16-byte value, exactly laid out on disk as

```
f055 4c49 4547 4549 4e46 4f30 3934 300f
```

The next values[2] are the number of blocks in the file system, the number of i-node blocks and the size of a block.

3.2 I-node

An i-node (a.k.a. file control block) is a 32-byte data structure depicted in Figure 3.

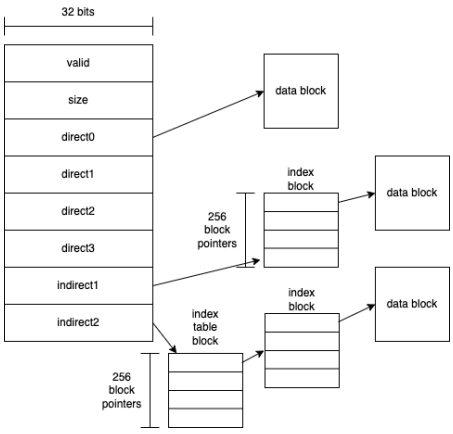

Figure 3: i-node.

The `valid` value is 0 if the i-node is not in use, and non-zero (use the value 1) if the i-node has been allocated to a file.

The `size` is the size of the file in bytes.

Then there are `direct` pointers[3] to the first four blocks of the file.

To support larger files, we then have two indirect pointers. The first indirect pointer, `indirect1`, points to an index block containing pointers to the next 256 blocks of the file.

[2] In SSFS, unless stated otherwise, all multi-byte values are little-endian. Most modern x86 CPUs use little-endian number representation, so you are probably fine not worrying about this – but do check!

[3] Pointers to blocks are simply the corresponding block number. The NULL pointer is encoded as 0.

The `indirect2` pointer points to a block containing an index table, where each entry is a pointer to an index block.

All i-nodes are packed in contiguous i-node blocks, as an array of i-nodes (see fig 1).

3.3 Allocation strategy

In SSFS, all allocations are done following a *first available* strategy: the allocated block or i-node will always be the one that is available and has the lowest number. You can think of the allocation strategy as always searching left-to-right in Figure 1.

4 SSFS API

Your SSFS implementation must exclusively use the virtual disk driver interface described in section 2 to access the virtual disk. You must implement the SSFS file system through the following interface:

```
int format(char *disk_name, int inodes);
int mount(char *disk_name);
int unmount();
int create();
int delete(int inode_num);
int stat(int inode_num);
int read(int inode_num, uint8_t *data, int len, int offset);
int write(int inode_num, uint8_t *data, int len, int offset);
```

Unless stated otherwise, all these functions return 0 on success and a negative integer on failure. You can use specific error codes or just return -1 for simplicity.

- `int format(char *disk_name, int inodes)` formats, that is, installs SSFS on, the virtual disk whose disk image is contained in file `disk_name` (as a c-style string). It will attempt to construct an SSFS instance with at least `inodes` i-nodes and a minimum of a single data block. `inodes` defaults to 1 if this argument is 0 or negative.

  Note that this function must refuse to format a mounted disk.

- `int mount(char *disk_name)` mounts the virtual disk, whose disk image is contained in file `disk_name` (as a c-style string), for use. Your implementation can safely assume that maximum a single volume with SSFS will be mounted at any given time, that is, `mount` should fail if it is called while another volume is already mounted.

- `int unmount()` unmounts the mounted volume. This can only fail if it is called when no volume has been mounted.

- `int create()` creates a file and, on success, returns the i-node number that identifies the file.

- `int delete(int inode_num)` deletes the file identified by `inode_num`.

- `int stat(int inode_num)` returns the file size on success.

- `int read(int inode_num, uint8_t *data, int len, int offset)` reads `len` bytes, from `offset` into file `inode_num`, into `data`. On success, it returns the number of bytes actually read.

- `int write(int inode_num, uint8_t *data, int len, int offset)` writes `len` bytes from `data`, at `offset` into file `inode_num`. If need be, any gap inside the file is filled with zeros. On success, it returns the number of bytes actually written from `data` (i.e. filling bytes are *not* counted in the return value).

4.1 Code organisation

Your code must follow the following partial organisation:

```
src
|---- Makefile
|---- fs_test
|---- include
|    |---- error.h
|    |---- fs.h
|    |---- vdisk.h
|---- main.c
|---- vdisk
|    |---- vdisk.c
```

You can organize the rest of your code as you see fit.

`make` compiles the application in `main.c` into `fs_test`.

We provide `vdisk.h`, `vdisk.c`, `fs.h`, `error.h` and `error.c` for you. You must *not* modify `vdisk.h`, `vdisk.c` and `fs.h`. You can, however, *add* to `error.h` and `error.c`.

We also provide some disk images to help you test your code. `DO NOT INCLUDED THESE IN YOUR SUBMISSION!`

5 Remarks

- `xxd` and `less` are your friends for this assignment. Learn to use them well. You may also want to take a look at `dd`.
- As our virtual disks are files and the operating system already does its magic to speed up file operations, there is no need for you to try and optimize operations on your file system implementation.
- Assume that a single threaded program will be using a disk image at any given time: do not try to make your implementation safe for concurrent use.
- You must be very careful to respect *scrupulously* the SSFS specification. Your code must be able to manipulate disk images created with somebody else's code, and somebody else's code must be able to manipulate disk images created with your code.
- Make sure to install `libbsd-dev` on the reference VM.

5.1 Code Requirements

The following requirements have to be satisfied:

- Your code **must** compile without errors and run without crashing on the reference VM (Ubuntu 24.04.1 LTS with kernel 6.8.0).
- You should also avoid having any warning during compilation.
- Your code must be readable. Use common naming conventions for variable names and **comments**.
- Your code must be robust and must not crash. In addition, errors handling and cleaning must be managed in a clean way.
- Your implementation **must** be in adequation with the defined interface.
- Only interact with the virtual disk through the provided virtual disk driver API.

6 Evaluation and tests

Your program can be tested on the submission platform. A set of automatic tests will allow you to check if your program satisfies the requirements. Depending on the tests, a **temporary** mark will be attributed to your work. Note that this mark does not represent the final mark. Indeed, other criteria such as the structure of your code, the memory management will be taken into account. You are however **reminded** that the platform is a **submission** platform, not a test platform.

7 Submission

Projects must be submitted through the submission platform before **Tuesday May 13$^{\text{th}}$, 23:59 CET**. Late submissions will be accepted, but will receive a penalty of $2^n - 1$ points (/20), where n is the number of days after the deadline (each day started counting as a full day).

You will submit a `src.tar.gz` archive of a `src` folder containing your code and `Makefile`. Failure to compile will result in an awarded mark of 0.

The submission platform will compile our own `main.c` and do basic checks on your submission, and you can submit multiple times, so check your submission early!

Bon travail...