CS60038: Assignment 2

Building a simple file system with an emulated disk, and interfacing with FUSE.

Assignment Date: October 18, 2019
Deadline: November 8, 2019

The objective of this assignment is to get hands-on experience in building a simple file system.

This assignment will involve building four parts:

- **PART A: Building a disk emulator**
- **PART B: Implementing a simple file system - SFS**
- **PART C: Implementing directory structure in SFS**
- **PARC D: FUSE interface for SFS**

PART A: Disk Emulator

Since it is difficult to test our file system on a physical raw disk, we will need to emulate it. For that we are going to create a file based disk emulator using standard open, read, write, and lseek calls in Linux. This will provide APIs for block level access to the emulated disk. Along with that, the disk will mantain some stats about its usage.

Our disk emulator interface will have the following functions:

```
int create_disk(char *filename, int nbytes);

int open_disk(char *filename);

disk_stat* get_disk_stat(int disk);

int read_block(int disk, int blocknr, void *block_data);

int write_block(int disk, int blocknr, void *block_data);

int close_disk(int disk);
```

The disk will have fixed block size of `4KB` (4019 bytes). In addition to providing the API for reading and writing blocks, it will maintain some stats about the disk.

The structure for the stats is as follows:

```
typedef struct disk {
    uint32_t size; // size of the entire disk file
    uint32_t blocks; // number of blocks
    uint32_t reads; // number of block reads performed
    uint32_t writes; // number of block writes performed
} disk_stat;
```

These stats will take up `16 bytes` of storage at the beginning of the disk.

Therefore, if we consider a disk of size `409600 bytes = (4KB x 100)`, the number of usable blocks will be `99`. Again, if we consider a disk of size `409616 bytes = (4KB x 100) + 16`, the number of usable blocks will be `100`.

Implement the disk emulator functions as follows:

`int create_disk(char *filename, int nbytes)`

Creates a disk file of size `nbytes`. Initializes `disk_stat` struct with calculated number of blocks, and reads, writes as 0. Writes it to the beginning of the disk. Returns 0 if successful, -1 for error. The size of the disk file must be exactly `nbytes`.

`int open_disk(char *filename)`

Opens the disk file of given filename. Check if valid stat is present. Return the **file descriptor** if successful. Return error otherwise. The returned file descriptor will be used in the following functions.

`disk_stat* get_disk_stat(int disk)`

Reads disk_stat from the disk. And returns its pointer (pointer to the structure disk_stat). This pointer can be used to access the disk stats.

`int read_block(int disk, int blocknr, void *block_data)`

Assumes that `block_data` is a 4KB buffer. Checks if `blocknr` is a valid block pointer - `blocknr` ranges from `0` to `(B -1)`, where B is the total number of blocks in the disk. Reads the block contents into the buffer. Returns 0 if successful, -1 otherwise. Also updates the `disk_stat` to increment the `reads` count.

`int write_block(int disk, int blocknr, void *block_data)`

Assumes that the `block_data` is a `4KB` buffer. Checks if `blocknr` is a valid block pointer. Writes the contents of the `block_data` buffer into the correct block. Returns 0 if success, -1 otherwise. Also, updates the `disk_stat` to increment the `writes` counter.
NOTE: Updating disk_stat does not affect the writes counter.

`int close_disk(int disk)`

Close the emulated disk file.

PART B: Simple File System SFS

Using the emulated disk we will be implementing a simple file system: SFS. Let us first take a look a the file system layout:

SFS will have the following components:

- Super Block
- Inodes and Inode Blocks
- Data Blocks
- Indirect Blocks
- Inode Bitmap
- Data block bitmap

Super Block

The first block of SFS will be the super block. The super block will have the following info:

```
typedef struct super_block {
        uint32_t magic_number;  // File system magic number
        uint32_t blocks;        // Number of blocks in file system (except super block)

        uint32_t inode_blocks;  // Number of blocks reserved for inodes == 10% of Blocks
        uint32_t inodes;        // Number of inodes in file system == length of inode bit map
        uint32_t inode_bitmap_block_idx;  // Block Number of the first inode bit map block
        uint32_t inode_block_idx;       // Block Number of the first inode block

        uint32_t data_block_bitmap_idx; // Block number of the first data bitmap block
        uint32_t data_block_idx;        // Block number of the first data block
        uint32_t data_blocks;  // Number of blocks reserved as data blocks
} super_block;
```

Inodes and Inode Blocks

```
typedef struct inode {
        uint32_t valid; // 0 if invalid
        uint32_t size; // logical size of the file
        uint32_t direct[5]; // direct data block pointer
        uint32_t indirect; // indirect pointer
} inode;
```

As we can see the size of each inode is `32 bytes`. Therefore each inode block will contain `128` inodes.

The number of blocks assigned for inodes will be `10%` of the total number of available blocks in the disk (except super block).

According to the total number of inodes, the length of the inode bit map is set.

Rest of the blocks are used for data blocks and bit map for data block.

Data Blocks

Data blocks are the blocks used for storing actual file contents. We can also use these blocks as indirect blocks for storing indirect block pointers. Later we will be using these data blocks for storing directory structures also.

Indirect Blocks

These are data blocks storing pointers to other data blocks. In case direct pointers are not enough to accomodate all the block pointers, the indirect blocks will be used to store more pointers. Each pointer is an integer of `4 bytes`, so one indirect block will hold `1024` such pointers. Only one indirect block is allowed per inode.

Inode Bitmap

The inode bitmap is used to determine which inodes are occupied and which are free. Accordingly while creating an inode, the first free inode index is determined from this bitmap.

Data block bitmap

Similar to inode bitmap, the data block bitmap is used to find out which blocks are free or used.

The following functions needs to be implemented for SFS:

```
int format(int disk);

int mount(int disk);

int create_file();

int remove_file(int inumber);

int stat(int inumber);

int read_i(int inumber, char *data, int length, int offset);

int write_i(int inumber, char *data, int length, int offset);
```

`int format(int disk)`

This function takes input the file descriptor of the emulated disk. In order to write the super block first we need to calculate all the values of blocks, indexes etc.. So the organization of the file system needs to be decided first. This is done as follows:

If the disk has `N` blocks (usable blocks as specified in `disk_stat`), one block is reserved for the super block. Therefore we have `M = N - 1` blocks remaining. Among these M blocks, 10 percent is reserved for inode blocks. Therefore the number of inode blocks is `I = 0.1 * M` (take floor). Therefore the total number of inodes will be `I * 128`, which is also the length of the inode bitmap (in bits). Therefore the number of blocks reserved for the inode bitmap will be `IB = (I * 128) / (8 * 4096)` (take ceil).

Therefore the remaining number of blocks is `R = M - I - IB`. These will be used as data blocks and data block bitmap blocks. For simplicity, consider that the length of the data block bitmap is `R bits`. Therefore the number of blocks required for data block bitmap is `DBB = R / (8 * 4096)` (take ceil). Therefore, the remaining are data blocks: `DB = R - DBB`. NOTE: we only consider the first `DB` number of bits of the data block bitmap.

The values of the super block structure will be set as follows:

```
magic_number = 12345
blocks = M
inode_blocks = I
inodes = I * 128
inode_bitmap_block_idx = 1
```

```
inode_block_idx = 1 + IB + DBB
data_block_bitmap_idx = 1 + IB
data_block_idx = 1 + IB + DBB + I
data_blocks = DB
```

The super block is then written to the disk.

The bitmaps are initialized. The inodes are also initialized with `valid = 0`. All these are written to the disk.

Returns 0 if successful, -1 otherwise.

`int mount(int disk)`

This method first attempts to read the super block and then verify the `magic_number`. If this is successful, a valid SFS is detected. In that case load the bitmaps as well as the mounted file descriptor into the memory (this file descriptor is used in the next function calls). Return 0 if successful, -1 otherwise. The except `format`, and `mount`, other functions should return error when operating on unmounted filesystem.

`int create_file()`

Creates a new inode and returns the inode pointer - that is the index of the inode (starting from 0). Initialize the size to 0 and valid to 1. Accordingly update the inode bitmap (not only in memory but also on the disk). Return -1 if error or if no inodes are left.

`int remove_file(int inumber)`

Removes an inode along with all corresponding data blocks and indirect pointer blocks. Just set valid to 0 in the iode, and update the bitmaps (both inode and data block bitmaps).(No need to override data blocks or indirect pointer blocks).

`int stat(int inumber)`

Returns the stats of the input inode: logical size, and number of data blocks in use, number of direct and indirect pointers in use.

`int read_i(int inumber, char *data, int length, int offset)`

First checks if `inumber`, `offset` and `length` are valid or not. In case the `inumber` is invalid or `offset` in out of range return error (`-1`). If length is out of range, read only till the end of the file and return the length of the data read (in bytes). The data is read into the buffer `data`, which is assumed to be of length `length`. In case of any error return `-1`.

`int write_i(int inumber, char *data, int length, int offset)`

First checks if `inumber` and `offset` are valid. Then write data to a valid inode by copying `length` bytes from the buffer `data` into the data blocks of the inode starting at offset bytes. Allocate any necessary direct and indirect blocks in the process. Also update the bitmaps accordingly. Returns the number of bytes actually written. In case of any error return `-1`. If the disk is full (no free data blocks left, write as much data as possible and return the number of bytes written).

PART C: File and directory structure in SFS

Now since we have the basic functions to create, remove, read and write files, we need build a directory structure to organize files. Use the functions above to create your own directory structure. Remember every directory is also a file, and the file contains a list of its children. The list should contain:

- valid bit (unset when a file or directory is deleted and can be replace by some other entry later)
- type bit: file / directory
- the name of the file or directory (fix the allowed maximum length)
- length of the name
- inumber of the file or directory

Assign the first inode for the root directory file.

You can use the following structure for the entries in the directory files:

You may modify it to include other information if required. (provide justification)

Implement the following additional functins to read and write files with file path:

```
int read_file(char *filepath, char *data, int length, int offset);
int write_file(char *filepath, char *data, int length, int offset);
int create_dir(char *dirpath);
int remove_dir(char *dirpath);
```

PART D: Interfacing with FUSE

FUSE (Filesystem in Userspace) is an interface for userspace programs to export a filesystem to the Linux kernel. FUSE is particularly useful for writing virtual file systems. Unlike traditional file systems that essentially save data to and retrieve data from disk, virtual filesystems do not actually store data themselves. They act as a view or translation of an existing file system or storage device.

libfuse: https://github.com/libfuse/libfuse

Documentation: http://libfuse.github.io/doxygen/

Tutorials:

- http://www.maastaar.net/fuse/linux/filesystem/c/2016/05/21/writing-a-simple-filesystem-using-fuse/
- http://gauss.ececs.uc.edu/Courses/c4029/labs/Fuse/index.html

NOTE: In all the functions, take special care to free any allocated memory that is not required once the function returns.

For bitmap manipulations you may follow this tutorial: http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html

Important Information

The students have to submit the source code compatible with the header files provided. A Makefile is also included for compiling the code. Implement the disk emulator code and SFS code in disc.c and sfs.c respectively. For FUSE you may create your own files and change the Makefile accordingly. Use main.c to test the codes.

In every source code file, students need to comment their name, roll number.

Do not make any assumption. If you find anything ambiguous in the assignment statement, do not hesitate to post it to the piazza.

DO NOT COPY CODE FROM OTHER GROUPS. WE WILL NOT EVALUATE A SOLUTION IF FOUND COPIED. ALL INVOLVED GROUPS WILL BE AWARDED ZERO.