IN2140 - Introduction to Operating Systems and Data Communications

Mandatory Assignment 2: Your Own File System

Spring 2022

1 Introduction

In this mandatory assignment, you will implement a sketch of your own file system. We call it a sketch because your solution obviously does not have to include all the functionality that a complete file system should have. If you later take the course IN3000/IN4000 - Operating Systems, then you will implement a more advanced file system solution. Much of what you learn in this assigment can come in handy. You are given a structure inspired by inodes on Unix-like operating systems, such as OpenBSD, Linux and MacOS. The structure contains metadata about files and directories, including pointers to other inodes in the file system. In this way, a tree with one root note is formed, as shown in the example below.

```
/
├── etc/
│   ├── httpd/
│   │   └── conf/
│   │       └── httpd.conf
│   ├── host.conf
│   └── hostname
└── var/
    └── log/
        └── messages
```

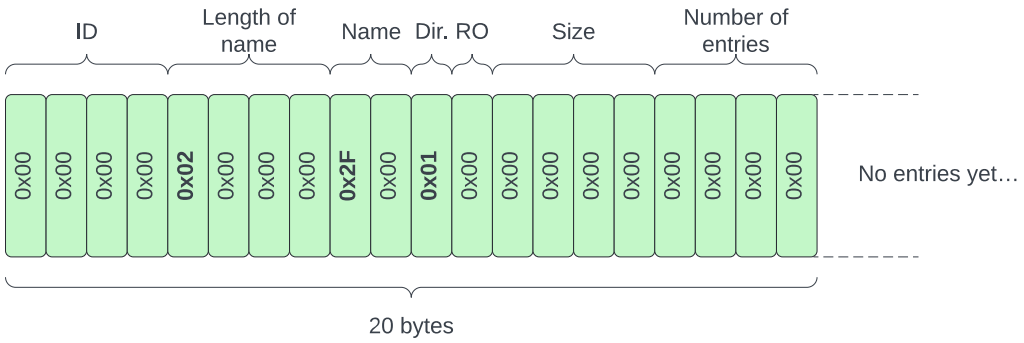
You will need to implement functions to work with the file system. Each file and each directory is represented by an inode, which has a type, a name, some other attributes and information about the contents of the file or directory. To be able to read and change inodes, they must be loaded from disk to memory. After changes are made, they must be written back to disk. When stored in memory, an inode has the following structure:

```
struct inode
{
    int     id;
    int     name_len;
    char*   name;
    char    is_directory;
    char    is_readonly;
    int     filesize;
    int     num_entries;
    size_t* entries;
};
```

In the following, we explain the fields in `struct inode`:

- Each inode has an ID number `id` unique to the entire file system and a file or directory name `name` with length `name_len`, which includes the final null byte. For example, the root directory will have a name `"/"` with a length of two bytes. Arbitrarily long file and directory names are allowed.
- The byte flag `is_directory` determines whether the inode represents a file or directory. If `is_directory` is null, then the inode represents a file.
- Another byte flag, `is_readonly` determines whether the file or directory that the inode represents is readable and writable or readable only. It is included as an example attribute, but has no practical significance for the assignment.
- `Filesize` returns the file size in bytes; for directories this is always zero.
- Each inode has a pointer `entries` to an array of `num_entries` entries of 64 bits. The entries are interpreted differently depending on whether the inode represents a file or a directory: For files, the entries are interpreted as block numbers (integer type), where each block has a size of 4096 bytes. For directories, each entry contains an inode pointer, which represents a file or subdirectory of that directory.

The figure below shows what the inode representing the root directory looks like when it is on disk. Note that the number of entries is zero as long as no other files or directories have been created. Later, each entry will fill 8 bytes. `0x2F` is, by the way, the hexadecimal ASCII value for forward slash.



2 Tasks

2.1 Design

Write a README file explaining the following points:

- How to read the superblock file from disk and load the inodes into memory.
- Any implementation requirements that are not met
- Any part of the implementation that deviates from the precode. For example, if you are creating your own files, explain what the purpose is.
- Any tests that fail and what you think the cause may be.

2.2 Implementation

We provide precode and expect you to implement the following four functions, which are found as skeletons in the file `inode.c`. It will be helpful to write help functions as well.

Create file

```
struct inode* create_file(
    struct inode* parent,
    char* name,
    char readonly,
    int size_in_bytes
);
```

The function takes as parameter a pointer to the inode of the directory that will contain the new file. Within this directory, the name must be unique. If there is a file with the same name there already, then the function should return `NULL` without doing anything.

Create directory

```
struct inode* create_dir(
    struct inode* parent,
    char* name
);
```

The function takes as parameter the inode of the directory that will contain the new directory. Within this directory, the name must be unique. If there is a directory with the same name there already, then the function should return `NULL` without doing anything.

Find inode by name

```
struct inode* find_inode_by_name(
    struct inode* parent,
    char* name
);
```

The function checks all inodes that are referenced directly from the parent node. If one of them has the name `name`, then the function returns its inode pointer. `Parent` must point to a directory inode. If no such inode exists, then the function `NULL` returns.

Load

```
struct inode* load_inodes();
```

The function reads the superblock file and creates in memory an inode for each corresponding entry in the file. The function puts pointers between the inodes correctly. The superblock file remains unchanged.

Shutdown

Type the following function to shut down the file system.

```
void fs_shutdown (struct inode * node);
```

The function frees all inode data structures and all memory to which they refer. This can be done just before a test program ends the program.

2.3 Important features provided

```
void debug_fs (struct inode* node);
```

The precode provides a function `debug_fs` which prints an inode and if this inode is a directory, recursively also all file and directory inodes under it. ID, name and additional information are written.

```
void debug_disk ();
```

In addition, the `debug_disk` function is provided, which prints either 0 or 1 for all blocks on our simulated disk. If a block has a value of 1, there is a file inode that has reserved the block for its file. Blocks represented by at 0 are not in use.

```
int allocate_block ();
```

The function allocates a disk block of 4096 bytes. You cannot allocate less than one disk block to file data, and the block can not be shared between files. The function returns a block index or -1 for no available blocks. We keep this information updated on disk in a file called `block\_allocation\_table`. This simple file system does not implement the extent principle. One must allocate one block of 4096 bytes at a time by calling the function. We assume that the superblock is not managed in the same way.

```
int free_block (int block);
```

When the file is deleted, the disk blocks must be released. Also returns -1 in case of error (block not allocated and so on).

3 Advice

Here are some tips to help you get started.

- It may be appropriate to implement the functions in this order:

1. `load_inodes`
2. `create_dir`,
3. `create_file`
4. `find_inode_by_name`
5. `fs_shutdown`.

- To check memory leaks, run Valgrind with the following flags:

```
valgrind
    --track-origins=yes   \
    --malloc-fill=0x40    \
    --free-fill=0x23      \
    --leak-check=full     \
    --show-leak-kinds=all \
    DITT_PROGRAM
```

4 Delivery

1. Put all the files in a folder with your username. Apart from Makefile, you do not need to create any new files yourself, but can manage with the files that you have received as a precode (after you have implemented the functions). If you create your own files, remember to copy them into the folder as well.

```
$ mkdir brukernavn
$ cp -r
        create_example1/ \
        create_example2/ \
        create_example3/ \
        load_example1/   \
        load_example2/   \
        load_example3/   \
        allocation.c     \
        allocation.h     \
        inode.c          \
        inode.h          \
        Makefile         \
        brukernavn/
```

2. Create a compressed archive, which you deliver.

```
$ tar czf username.tar.gz username/
```

3. Highly recommended! Test the submission by downloading the tar.gz file to an Ifi machine. Unpack, compile and run.