IN2140—Introduction to Operating Systems and Data Communications

Mandatory Assignment: Your Own File System

Spring 2024

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 node 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.

In memory

When stored in memory, an inode has the following structure:

```
struct inode
{
    int             id;
    char*           name;
    char            is_directory;
    int             num_children;
    struct inode** children;
    int             filesize;
    int             num_blocks;
    size_t*         blocks;
};
```

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

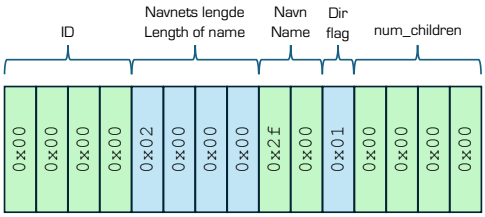
- Each inode has an ID number `id` unique to the entire file system and a file or directory name `name`, which includes the final null byte. The root directory will always have the name `"/"`. Arbitrarily long file and directory names are allowed. The length of the name, including the final null byte, is stored on disk followed by the name itself. Storing the name length on disk makes your work a little easier.
- The byte flag `is_directory` determines whether the inode represents a file or directory. If `is_directory` is 0, then the inode represents a file.
- If the `is_directory==1`, the inode represents a directory that can contain both other directories or files. The number of files and directories is stored in `num_children`. If it is 0, `children` is NULL. Otherwise, `children` contains a pointer to a dynamically allocated array of inode pointers, which point to the children. Only the first `num_children` pointers are valid.
- If the `is_directory==0`, the inode represents a file. This file has the `filesize`, giving a simulated file size in bytes. The file has also `num_blocks`, which is the number of (simulated) blocks required to store the file with `filesize` bytes. It is computed by rounding up filesize/4096.
- To track where these blocks are stored on the (simulated) disk, there is `blocks`. This is a pointer to a dynamically allocated array big enough to hold `num_blocks` `size_t` values that represent the number of a disk block. `size_t` is an unsigned integer value, which is 8 bytes large on the x86/Linux computers at IFI (see `man size_t` for more information).

On disk

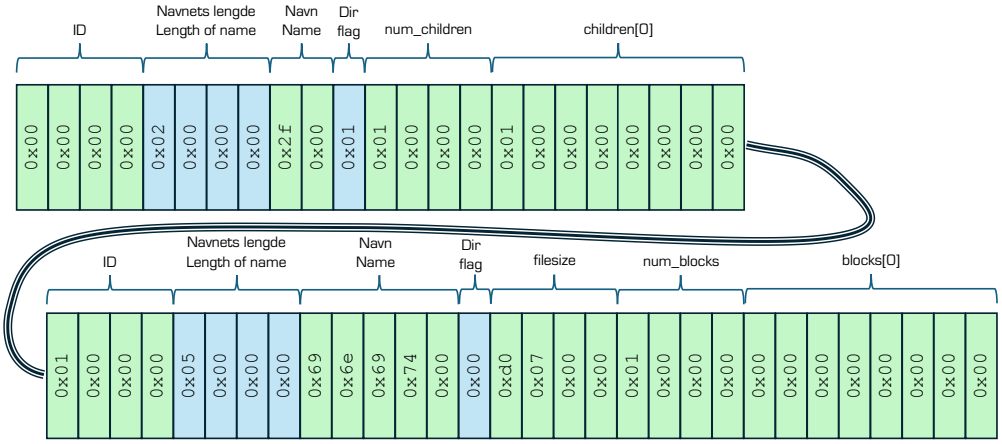
Not all of these fields are stored in the master file table on disk. The most important detail is that `children` are memory addresses (pointers) that make no sense when they are stored on disk. We store the children's ID instead.

Figure 1 shows what the master file table looks like on a simulated disk that contains only the empty root directory. Note that the `num_children` is 0 as long as no other files or directories have been created. Later, each entry will fill 8 bytes.

Figure 2 shows a master file table for a simulated disk containing the root directory '/' and a single file called 'init'. You can see that information about children is only stored for directories, and that information about file size and the blocks that are used on the simulated disk is only stored for files. The entry `children[0]` contains **the ID** of the inode for 'init' (and not a pointer to it).



Figur 1: Master file table for a formatted but empty filesystem, containing only the root directory '/'. Every cell represents a byte on disk. Note that the integer values are stored in little endian byte order.



Figur 2: Master file table for a formatted filesystem, containing only the root directory '/' and the file 'init'.

2 Tasks

2.1 Design

Write a README file explaining the following points:

- How to read the master file table (as described in the lecture slides) 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 their 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 functions, which are found as skeletons in the file `inode.c`. It will be helpful to write helper functions as well.

Create File

```
struct inode* create_file(
    struct inode* parent,
    char* name,
    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 or directory with the same name there already, then the function should return `NULL` without doing anything.

The parameter `size_in_bytes` gives the number of bytes that must be stored on the simulated disk for this file. The necessary number of blocks must allocated using the function `allocate_block`, which is implemented in allocation.c. It is possible that there is not enough space on the simulated disk, meaning that a call to `allocate_block` will fail. You should release all resources in that case and return `NULL`.

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 file or directory with the same name there already, then the function should return `NULL` without doing anything.

Delete a file

```
int delete_file(
    struct inode* parent,
```

```
    struct inode* node
);
```

The function takes as parameters the inodes of a directory and a file. The file, node, should be deleted. The directory, parent, contains this file. The inode node can be deleted by this call if parent is the direct parent of node. The delete_file operation fails if this is not the case. In case of success, delete_file returns 0, otherwise -1. The blocks allocated on the simulated disk must be released using the function `free_block`, which is implemented in allocation.c.

Delete a directory

```
int delete_dir(
    struct inode* parent,
    struct inode* node
);
```

The function takes as parameters the inodes of two directories. The second one, node, is the directory that should be deleted. The first one, parent, contains this directory. The inode node can be deleted by this call if parent is its direct parent, and if the directory node is empty (does not contain files or other directories). The delete_dir operation fails if this is not the case. In case of success, delete_dir returns 0, otherwise -1.

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 inode. 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 returns `NULL`.

Load File System

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

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

If the loading operation succeeds, the inode returned by this function should always be the root directory, meaning that its `name` field should point to the string "/".

Shut Down 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 save_inodes( const char* master_file_table, struct inode* root )
```

The precode provides a function `save_inodes`, which saves the master file table to disk. While the function `load_inodes` that you must write, is more difficult because you have to make context-depended decisions and have to manage dynamic memory correctly, this can help you a lot in understand how inodes in memory are transformed into inodes on disk.

```
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 master file table 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 advice to help you get started.

- It may be appropriate to implement the functions in this order:
  1. Functions required for loading:
     - `load_inodes`
     - `find_inode_by_name`
  2. Functions required to create files and directories and update the master file table and the block allocation table.
     - `create_dir`
     - `create_file`
     - `fs_shutdown`
  3. Functions required to delete files and directories:
     - `delete_file`
     - `delete_dir`
- 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 \
    YOUR_PROGRAM
```

4 Delivery

1. Put all the files in a folder YOUR_USER_IDS (all of your userids). 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 YOUR_USER_IDS
   $ cp -r
           create_example1/ \
           create_example2/ \
           create_example3/ \
           load_example1/   \
           load_example2/   \
           load_example3/   \
           del_example1/    \
           del_example2/    \
           del_example3/    \
           allocation.c     \
           allocation.h     \
           create_fs_1.c    \
           create_fs_2.c    \
           create_fs_3.c    \
           load_fs.c        \
           del_fs.c         \
           inode.c          \
           inode.h          \
           Makefile         \
           YOUR_USER_IDS
   ```

2. Create an archive that is compressed with the GNU tar program using the following command, which you deliver.

   ```
   $ tar czf YOUR_USER_IDS.tar.gz YOUR_USER_IDS/
   ```

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