IN2140—Introduction to Operating Systems and Data Communications

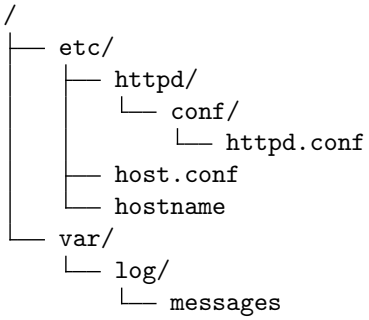
Mandatory Assignment 2: Your Own File System

Spring 2025

Innhold

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.



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 it contains enough information to find the data contained in that file on the physical disk itself. To be able to read and change inodes, they must be loaded from disk to memory. When stored in memory, an inode has the following structure:

1.1 `struct inode` in memory

```
struct inode
{
    uint32_t   id;
    char*      name;
    char       is_directory;
    char       is_readonly;
    uint32_t   filesize;
    uint32_t   num_entries;
    uintptr_t* entries;
};
```

In the following, we explain the fields in `struct inode` when it is loaded into memory:

- Each inode has an ID number `id` unique to the entire file system.
- An inode for a file or directory has a name `name`, which is a pointer to a C string. In general, those names can contain regular characters and numbers, as well as `'_'` and `'.'`. We do not allow whitespace (neither space nor tab). As a special case, the root directory has the name `'/'`. This root directory exists when the disk is freshly created (or formatted). 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 1, then the inode represents a directory. If `is_directory` is 0, then the inode represents a file. All other values are errors.
- 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` gives the simulated file's size in **bytes**; for directories this is always 0.
- Each inode contains a pointer `entries` to a dynamically allocated 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, each entry is a pair of two unsigned 32-bit integers **blockno** and **extent**. The blockno is interpreted as block number of a block of 4096 bytes on disk. The extent is the number of consecutive blocks on disk starting with blockno, Only the values 1, 2, 3 and 4 are possible. There is a relationship between `filesize` and the number of blocks required. The required number of blocks is computed by rounding up filesize/4096.
  - For directories, each entry contains an inode pointer, which represents a file or subdirectory inside this directory.

1.2 Inode on disk

Inodes must stored on disk so the computer and file system can shut down and restart later. When the `struct inode`s are loaded back into memory at a later time, they will be created at different addresses in memory. It does therefore not make sense to store pointers like `name` and `entries`.

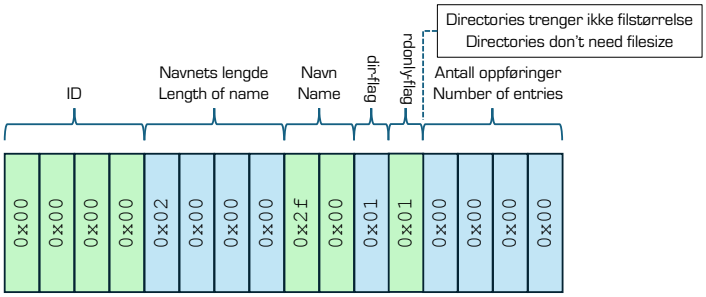
It is required to *serialize* the `struct inode`s when they are written to disk.

Here are some notes about this serialization:

- `struct inode`s on disk have different sizes, and several reads will be required to read a single `struct inode` from disk into memory.
- 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 will make the work a little easier.
- If an inode is a directory and it is serialized, the `entries` are written as to disk as a sequence of 64-bit-field that contain the `id`s of the `struct inode` that the entry in memory points to.
- If an inode is a file and it is serialized, the `entries` are written as to disk as a sequence of 32-bit-field pairs. The values (blockno and extent) are the same as in memory.

Figure 1 below shows what the inode representing the empty root directory looks like when it is on disk. Note that the number of entries is 0 as long as no other files or directories have been created. Later, each entry will fill 8 bytes.

You can see that the 'name' field contains `0x2F` and `0x00`, which are the hexadecimal ASCII value for forward slash '/' followed by the string-terminating zero.



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.

Figure 2 shows the top-level directory '/' when it contains a first file named 'init'.

You can see that the inode for '/' takes more space now. The entry in this directory inode now contains **the ID** of another inode (and not a pointer to it).

The inode for 'init' contains 4 bytes to store the file size, and it also has one entry. This entry comprises two deler, blockno and extent. When you look at the example, you see that filesize contains 0x07d0 in hexadecimal, which is 2000 in decimal.

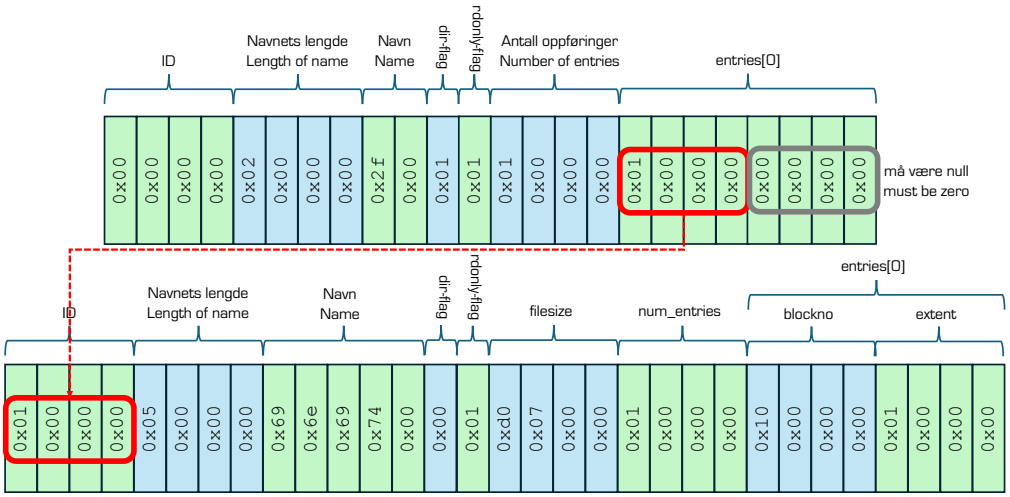
Every blocks on the disk is 4096 bytes big, so 'init' fits into a single block. Consequently, there is only one entry, it is the block 0x10 (decimal 16) and the extent is 1.

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.



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

- 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,
                           const const 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 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,
                          const 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.

Find Inode by Name

```
struct inode* find_inode_by_name( struct inode* parent,
                                  const 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`.

Delete a file

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

This functions deletes a file referred to by its inode `node` from its parent directory referred by its inode `parent`.

The function calls `free_block` for every block that is referenced by this file. This applies also to extents: if the extent has the length of 4, `free_block` must be called 4 times. This removes those blocks from simulate disk.

This function does not do anything and returns an error if `node` is not a file, if `parent` is not a directory, or if `parent` is not the directory that contains `node`.

Returns 0 on success and -1 on error.

Delete a directory

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

This functions deletes an empty directory referred to by its inode `node` from its parent directory referred by its inode `parent`. It releases all memory associated with `node`.

This function does not do anything and returns an error if `node` is not a directory, if `parent` is not a directory, or if `parent` is not the directory that contains `node`. It also does nothing and returns an error if `node` is a directory but is not empty, ie. still contains other inodes.

Returns 0 on success and -1 on error.

Save File System

```
void save_inodes( const char*   master_file_table,
                  struct inode* root );
```

The functions writes the given inode root and all inodes referenced by it to the master file table, following the oblig instructions. No inodes are changed.

Load File System

```
struct inode* load_inodes( const char* master_file_table );
```

The function reads the given 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 remains unchanged.

Returns a pointer to the root inode on success and `NULL` on error.

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 Discussion with evaluator

During the evaluation of the mandatory assignment, evaluators will randonly select groups that must be prepared meet their evaluator for a discussion.

A group that is asked by their evaluator to meet for such a discussion must do so, it is not optional or open for negotiation. The discussions will be quite short and should preferably happen during one of the group sessions. The evaluator must be confident that each group member has understood the code that you have delivered.

The invitation for discussion may come whether the evaluator believes that a solution is ready to pass without further changes or not. Going broad will help us to get a better understanding of the courses participants readiness for the home exam.

The discussion will usually replace the written feedback for the groups that are invited for a discussion (although the evaluator is allowed to write as well), so we advise groups to take notes carefully.

2.4 Expectations

The mandatory assignment can be passed when all examples that are only reading the master file table and block allocation table are working flawlessly, meaning that it is possible to pass without even attempting any of the operations that change the tree of inodes in memory.

However, we interpret flawlessly very narrow here: no compiler warning after making in a freshly created build directory, not runtime warnings, no warnings or errors from valgrind, no crashes, and the expected output is matched perfectly for the tests that we hand out as well as for the additional tests that we are not handing out.

To compensate for shortcoming, you must make progress towards the correct working of the modifying and writing operations as well.

3 Further details

3.1 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.

If this function crashes or valgrind discovers out-of-bounds reading when you try to debug your inodes, it is very likely that you have not implemented inodes as intended by this assignment.

```
void debug_disk();
```

`debug_disk` prints a several lines of 0s and 1s. Every 0 or 1 represents a block on the disk, starting with block 0. 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.

If you (for example) suspect mistakes in allocating and freeing extents, you can compare the block numbers stored in your inodes with the blocks numbers shown as allocated by this function.

```
int allocate_blocks( int extent_size );
```

The function attempts to allocate up to `extent_size` consecutive disk blocks, where each block can holds 4096 bytes. You cannot allocate less than one disk block for any file, and a block cannot be shared between files. The function returns the block index of the first block in the extent or -1 if the request cannot be fulfilled completely. If an allocation fails, you should retry with a smaller `extent_size`. It will always fail when $extent\_size <= 0$ or $extent\_size > 4$. We keep this information updated on disk in a file called `block_allocation_table`.

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

When the file is deleted, the disk blocks must be released. It is important that every single block of an extent must be released separately becayse the `block_allocation_table` knows nothing about extents. `free_block` returns -1 in case of error (block not allocated and so on).

3.2 Advice

Here are some advice to help you get started.

- It may be appropriate to implement the functions in this order:
  1. `load_inodes` and `find_inode_by_name`
  2. `create_dir` and `create_file`, allocate only extents of size 1 at this time
  3. `fs_shutdown`
  4. `delete_dir` and `delete_file`
  5. add support for extents of size 2, 3 and 4 in `create_file` and `delete_file`
- 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
```

3.3 Compilation

To compile the programs, we have created a specification file called CMakeLists.txt again.

You can download and compile this project as follows: Open a terminal.

- In the terminal, unpack the provided files:

  ```
  tar zxvf oblig-02.tgz
  ```

- go into the oblig-02 directory

  ```
  cd oblig-02
  ```

- create a subdirectory to build the programs from the provided source code

  ```
  mkdir build
  ```

- go into the build directory

  ```
  cd build
  ```

- use CMake to create a Makefile that suits your machine

  ```
  cmake ..
  ```

- compile to create executable files from the wedge code

  ```
  make
  ```

- run the programs, e.g.

  ```
  create_example1
  ```

3.4 Testing

For your convenience, we are provided several test programs that test subsets of the functions that you must implement. The test are the following:

- `check_disk`: This is only test the block allocation functions that have been handed out. If the output of this test is not the same as the expected-outputs/test-1-1-expected-output.txt, something is wrong with your setup.
  NOTE: When we say 'the same', we don't mean literally the same, because the test output contains process numbers that are always different, as well as the tester's home directory name, which is different for each of us. But the output of the program itself, as well as the test results from valgrind should be the same.

- `check_fs`: This loads the given master file table and block allocation table into memory, prints the directory tree with additional information for each inode, prints a map of the blocks that are allocated according to the information stored in the inodes, and it prints a map of the blocks that are actually allocated according to the block allocation table. The expect results are in expected-outputs/test-2-*-expected-output.txt

- `load_fs_1`, `load_fs_2`, and `load_fs_3`: Starts like `check_fs` but are meant to load specific example tables. Tries to find specific files and directories inside those loaded file systems. The expect results are in expected-outputs/test-3-*-expected-output.txt

- `create_fs_1`, `create_fs_2`, and `create_fs_3`: Starts with empty master file table and block allocation table that are given on the command line, formats the disk,

and creates files and directories are specified in the source code of these 3 test commands. The expect results are in expected-outputs/test-4-*-expected-output.txt

- `create_and_delete_test`: Like `create_fs_*`, but additionally, tries to perform some deletes of files and directories that are meant for fail, and some deletes that are meant to succeed. The expect results are in expected-outputs/test-5-1-expected-output.txt

4 Delivery

You should submit your group's code to `https://devilry.ifi.uio.no/` as an archive file. The archive file should be a Zip file and must contain all source files and documentation files.

If you follow the 'make' recipe under Compilation, you can create the archive file by calling

```
make package_source
```

The file is then named Oblig02-1.0-Source.zip.

It is highly recommended that you test your submission before uploading it! Test the submission by downloading the tar.gz file to an Ifi machine. Unpack, compile and run.