Robert Alvarez, Emiliano De La Cruz, & Diana Garcia
HW3 – December 6th, 2022

Overall Project

SWE decisions:

We make the following structs to make our list of free objects: (Memory allocation)

```
typedef struct s_AllocateFrom {
  size_t remaining;
  __myfs_off_t next_space;
} AllocateFrom;
typedef struct s_List {
  __myfs_off_t first_space;
} List;
```

The following structs are use to structure our file system: (FUSE)

```
typedef struct __handler_t {
   uint32_t magic;
   __myfs_off_t root_dir;
   __myfs_off_t free_memory;
   size_t size;
 } handler_t;
typedef struct __file_block_t {
   size_t size;
   size_t allocated;
   __myfs_off_t data;
   __myfs_off_t next_file_block;
  } file_block_t;
  typedef struct __inode_file_t {
   size_t total_size;
   __myfs_off_t first_file_block;  //This is an offset to the first file_block_t
  } file_t;
  typedef struct __inode_directory_t {
    //Max number of children is given at the children array location - sizeof(size_t) divided by the sizeof(__myfs_off_t)
    size_t number_children;
    //children is an offset to an array of offsets to folders and files. Children starts with '..' offsets
    __myfs_off_t children;
} directory_t;
typedef struct __inode_t {
  char name[NAME_MAX_LEN + ((size_t) 1)];
  char is_file;
  struct timespec times[2]; //times[0]: last access date, times[1]: last modification date
  union {
    file_t file;
```

```
    directory_t directory;
  } type;
} node_t;
```

Our file system starts by making a header_t struct to see if the filesystem is mount or remount (magic), an offset to the root directory (root_dir), an offset to the first AllocateFrom object (free_memory), and a size variable with the number of bytes that the file system was given to use.

Problems encounter:
The main issue was that I set by hand the root directory attributes byte by byte which was hard but it help to understand the offsets and pointer a lot to continue with the project.

Helper functions (memory allocation)

Add_allocation_space: Given where the file system starts, we can get the list of free block so we can add the allocation block to it. We use a similar approach to merge block as in HW2.
Extend_pref_block: This function extend a given block up to the size bytes required if the block that follows it is found withing the available free blocks.
Get_allocation: This function extends a prefer block (if any pass) and if the extension of the block don't get as many bytes as the ones requested it returns a block with all or as many found.
__malloc_impl: It behaves as the regular malloc but it gets the blocks from the list of free blocks that the file system has (it calls get_allocation).
__realloc_impl: Similar as __malloc_impl but this time we call get_allocation without a block of preference so we can return a block with everything that is needed (this function is mainly use to reallocate directories to keep them in a single block)
__free_impl: Add the block back into the list of free blocks. It calls add_allocation_space.

Helper functions (FUSE)

Off_to_ptr: Given an offset, a pointer to where is memory the offset is "pointing to" is return.
Ptr_to_off: Given a pointer, an offset is return base on the starting point of the file system.
Update_time: Given a node (inode) and a number between 0 and 1 we update the node times. If 0 is given, we only update the last access time. Otherwise, we update last access and modification time.
Get_free_memory_ptr: Given the file system starting point, we return a List object that have an offset to the first free memory space in our file system.
Handler: Given where the file system starts and the size of it, we check if the file system has been mount for the first so we set it up or we do nothing.
Get_last_token: Given a "string" with the path and a pointer to an unsigned long, we traverse the path from right to left until we find a '/' where we stop traversing the path,

we proceed to make a copy of it to be return and set the unsigned length to the length of the token.
Tokenize: Given a token, a path, and how many tokens we must skep, we tokenize the path using the token and return an array of “strings”.
Free_tokens: It take the return value of tokenize and call free for each of the tokens and the array itself.
Get_node: Given where the file system starts, a children dictionary (array of offset to get to the children of a folder), and a child name, we traverse the childs on the dictionary and return a pointer to the node that correspond to the child’s name, or NULL if name didn’t exist on the dictionary.
Path_sover: Given where the file system starts, a path and how many tokens to use to get to the corresponding node of interest, we use tokenize and get node get to it and return it upon success or NULL otherwise.
Make_inode: Given where the file system starts, a path, a pointer to set the error number and if the node to be created is a node or not, we traverse the path using path_solver by skiping the last node (which must be a folder), and use get_last_token to get the name of the node to create. We proceed to check that a node doesn’t exist with that name, check that the node name is valid, make room is needed on the parent folder to add a new offset to get to the new node. Once this has been check and modify as need we continue to call __malloc_impl to create the node_t object and set its characteristics base on the information if it was going to be a file or a folder.
Free_file_info: Given where the file system starts and a file_t object, we traverse over the file_blocks that it contains to free the data and file blocks.
Remove_node: Given where the file system starts, a children dictionary of offsets and a node, we traverse the directory until an offset match the one of the node given and we proceed to free the node and update the children offset by one to the left for the once remaining and update the number of children.
Remove_data: Given where the file system starts, the first file_block_t from a file node and the number of bytes to be skep, we traverse the block until we get to the block the first byte than needs to be removed and we start freeing all data blocks and blocks after that byte.
Add_data: Given the starting point of the file system, a file_t, the number of bytes to add and a pointer to set the error number if needed, we traverse the number of bytes indicated by the parameter and start calling __malloc_impl to extend the file blocks information until we get the number of bytes requested or fail to do so.

FUSE functions

__myfs_getattr_impl()
SWE decisions:

We call handler to set up the file system if needed. Call path_solver with the path given and check that the return node is valid otherwise we fail. If node is valid, we populate stbuf which for the file is straight forward because we can use the node->type.file.total_size as well as node->times[0] and 1 to easily populate

the stbuf. If the node is a folder, we need to traverse the children offsets to check how many folders are there.

Problems encounter:

When the node was folder we need to traverse the children starting at index 1 because index 0 have the parent offset and "." don't exist as an offset. Also, the count start at 2 because of "." and "..".

__myfs_readdir_impl()

SWE decisions:

We call handler to set up the file system if needed. Call path_solver with the path given and check that the return node is a directory otherwise we fail. Once we confirm it is a directory, we make space with calloc for an array of number of children (except ".." and ".") entries pointers to char. Once we have the space, we iterate over all children in the directory and make a copy on malloc memory space of its name and make an entry of the previous array to point to that copy. Once all children name had been copy we make namesptr to point to the array made by calloc and return the number of entries on it.

Problems encounter:

The main issue was to keep track of where to allocate and use the pointers to accurately save the names and null terminate them.

__myfs_mknod_impl()

SWE decisions:

This function directly calls make_inode which handles everything that this function needs to do.

Problems encounter:

Main problem wasn't directly assign to this function in specific but to get __malloc_impl and __realloc_impl to work the hard part to allocate the spaces needed to create our node. A particular of this problem was to set the space for 4 children offset if the node was a directory and to remember to set the first space as the offset to the parent.

__myfs_unlink_impl()

SWE decisions:

This calls path_solver skipping the last token so we have access to the parent folder, we also call get last token with get node to get the file node. Once we have both we call free_file_info to free all blocks and file nodes and then we proceed to call remove_node giving the parent directory and the file node.

Problems encounter:

We use a lot of helper functions which make this function lengthy and hard to follow when it was getting develop. Once everything was broken into small sections it became easy to follow. The hardest part was the freeing portions because we must keep track of the current and next block so we don't lose track of the files pointers.

__myfs_rmdir_impl()

SWE decisions:

We call path solver to get the directory that must be removed. We got the parent by converting the first child offset (the parent) into a pointer. We make sure that the directory of interest is empty to proceed. If empty, we free the children offsets array and then we call the remove_node with the parent children and the directory node.

Problems encounter:

At this point some other functions were created so no problems were encounter to this function in particular.

__myfs_mkdir_impl()

SWE decisions:

This function calls make_inode with a 1 instead of a 0 to indicate that it must create a directory instead of a file.

Problems encounter:

Same problems to bevelop make_inode as __myfs_mknod_implem().

__myfs_rename_impl()

SWE decisions:

This function was the only one that we didn't had time to implemented.

Problems encounter:

N/A

__myfs_truncate_impl()

SWE decisions:

After calling path solver we check if the new size is the same, so we just update last access time. If bytes need to be removed, we call remove_data with the number of bytes that would be keep so everything else gets free. If more bytes need to be added, we call add_data. If everything was successful, we update the new file total_size and the times of last access and editing.

Problems encounter:

The new function that was develop specifically for this function is add_data. This was probably the hardest because the files that get send to be extend may or may not had been initialize already. We also need to make sure that we can extend the block instead of just keep appending more into it. Lastly, if it fails because not enough bytes were collected to get the new size we must free the blocks starting at the last original byte. Lastly, the hardest part was to automate the new blocks appending until the desire size was reach or we fail.

__myfs_open_impl()

SWE decisions:

We call path_solver to get the node we want to open. If we successfully retrieve it we return if the node is a file or not.

Problems encounter:

No problems were encounter for this function.

__myfs_read_impl()

SWE decisions:

We call path_solver to get the node of interest, if the path was not valid or if the node is a folder we fail. Once we have the file node, we check that the offset is not outside the file size range, and we fail. If the file size is 0, we don't try to read so we terminate with success. Otherwise, we iterate over the file blocks until we reach the last byte that needs to be skip based on the offset and start reading from there until we get to the end of the file, or we fully populate the buffer.

Problems encounter:

Main issue was the amount of edge cases that needs to be taken care of. Also, iterating over the blocks of memory to skip the offset bytes was interesting to follow because this was the first function that we implement that needs to iterate over them and start at a not 0 offset.

__myfs_write_impl()

SWE decisions:

We start like truncate to get the file, check that is a file and append bytes if they are needed to be added. Then we jump offset number of bytes to see where we would start writing. We use a swapping approach of iterating throw the buffer and swap the current file character for the one on the buffer and we continue doing that until we reach the null character.

Problems encounter:

Our first issue was that the buffer was of type read only so we need to make a copy of it so we could use the swap approach mentioned above. Once that was done, we use a goto because the swap required a nested for loop, one to update the buf_idx when we got to the end and the other one to check that we are not on the last character, and we still have blocks so we must continue copying them. So, we goto serve us to exist both while loops from the inner one.

__myfs_utimens_impl()

SWE decisions:

We call path_solver to get the node that needs to be time updated. If the node was retrieved successfully, we just set times attribute of our node to the once passed on the parameter.

Problems encounter:

At the beginning we have our own times struct as the MS-DOS so once we saw it was easier to just use the premade struct timespec structure we change it for it instead.

__myfs_statfs_impl()

SWE decisions:

We set the block size to be of 1024 bytes and the maximum number of characters for a node name to be 255 bytes. Free block were calculated by traversing over the free blocks list, adding their bytes and dividing the total number by the size of a block.

Problems encounter:

No new issues were encounter on this function.

Testing

We make a main.c file for early testing on some helper functions like tokenize and get_last_token. Once those work we simulate a FUSE by calling mmap and some FUSE dependent helper functions like get_free_memory_ptr, __malloc_impl, and free_impl. We test those function by drawing the FUSE byte allocation base on the struct sizes and the order we implement for their allocation. The initial setup the file system is first mounted is as follows:

First 0x20 bytes where use for the header, next 0x0138 for the root node, next 0x28 for the block header and 4 children offset for the root directory children, and finally everything else was the first block and only (currently) block on the free blocks list. After all of that was multi check that it was correct, we proceed to test our file system by running it on the ssh that Dr. Lauter provide us with. Some operations where to touch a file, truncate the file to some number of bytes a couple times into higher and lower numbers that the current one. Make directories, make more files and folders inside of it by using touch and echo “<Insert something here>” > or >> <filename> and them removing everything using rm * or a variation of it to fit our testing objective. Then we also use ls, ls -l, and ls -la to make sure that every node that supposed to be (or not be) there was accurate. Lastly, we use vim <filename>, head, tail and cat to make sure that the read implementation work.