

# Assignment 3: A Memory File System

[Start Assignment](#)

**Due** Friday by 11:59pm      **Points** 100      **Submitting** a file upload

In this assignment, you will implement some operations for a toy in-memory File System, **MemFS**. This File System is very similar to the one we saw in class (Chapter 40: File System Implementation), but it keeps all blocks in memory, not in a storage device.

All the File System definitions are in a header file, `memfs.h`. It works as following:

MemFS is made of a collection of contiguous blocks, each one of 512 bytes in size. These blocks are not in a storage device, but in user memory. The number of blocks is configurable when instantiating the File System.

In MemFS there are two types of files, either regular ones (text files, images, etc), or directories (which can contain other files). Both types of files are represented by a struct called a *mem\_inode*. This struct contains information about the type of file, its last modification time, and its contents. Contents of a file are stored in other blocks. For regular files, the content blocks will contain text, data, etc depending on the type of file, and for directories, the blocks contain directory entries. There are a set of pre-allocated blocks that contain all the inodes in the system, and we can identify those inodes by a number.

Here is the definition for a `mem_inode`:

```
struct mem_inode {
    char type;                // File type
    char num_links;           // Number of links to inode in file system
    unsigned short size;      // Size of file (bytes)
    block_ref indirect_ptrs[NINDIRECT]; // Indirect data block addresses
    block_ref direct_ptrs[NDIRECT];    // Direct data block addresses
    unsigned int mtime;       // Last modified time
};
```

Block 0 in MemFS is always the File System's Superblock. Inside this block, we store metadata about the current system, such as the number of blocks, a magic number, etc.

```
struct superblock {
    int size;                // Size of file system image (blocks)
    int magic_number;        // Filesystem magic number
    int num_data_blocks;     // Number of data blocks
    int num_inode_blocks;    // Number of inode blocks.
    int num_inodes;          // Number of inodes
    int inode_start;         // Block number of first inode block
    int block_start;         // Block number of first data block
};
```

Blocks 1 and 2 are the Inode Bitmap and the Block Bitmap, respectively. A bitmap is a collection of bits that tell us if an inode (in the case of the Inode Bitmap) or a block (in the case of the Block Bitmap) are used. A 0 means the inode / block are not used, a 1 means they are used.

As mentioned above, `mem_inodes` have the information about their files stored as pointers to data blocks. Data Blocks can be found from a number by adding a number of bytes to the start address of the first data block. There are two types of block pointers in inodes, direct pointers (they contain the data block number where the contents are) or indirect pointers (they contain a block reference that in turn contains a list of additional blocks).

Directory inodes store directory entries (*mem\_dirents*) in their data blocks. Each entry consists of an inode number and a file name. In other words, only a directory inode knows about the names of its children. This is how a *mem\_dirent* looks like:

```
struct mem_dirent {  
    inode_ref inum;  
    char name[DIRSIZ];  
};
```

Based on this information, you need to implement a set of functions that perform operations on the MemFS. You can write them in a file, *memfs.c*, and test it from your own main driver. These are the functions you need to write:

### 1. ***void \*makefs(int num\_inode\_blocks, int num\_data\_blocks)***

This function creates the in-memory filesystem. It needs to first allocate enough memory to store all the blocks (Including the superblock, the two bitmap blocks, and the requested inode and data blocks.) This can be done with the *malloc()* or *calloc()* C functions. Once the memory has been allocated, the function needs to initialize the superblock in the first 512 bytes of the space. The values in *num\_inode\_blocks* and *num\_data\_blocks* are variable, but you can consider 32 for the first and 512 for the second as examples.

### 2. ***inode\_ref find\_free\_inode(void \*fs)***

This function goes through the bits in the inode bitmap (the second block in the File System) and finds the first position that is 0 (starting from position 1, since position 0 is reserved). It then returns that position. For example, if first byte of the inode bitmap was 01110010, this function would return 4.

### 3. ***block\_ref find\_free\_block(void \*fs)***

Same as the previous function, but returns the first position in the data block bitmap that is 0.

### 4. ***void create\_root\_dir(void \*fs)***

This function takes a MemFS created with *makefs()* and creates the root directory. The root directory of MemFS is called / (slash) and is a special directory because it is always located in the same inode

number (defined by `ROOT_INUM` in `memfs.h`.) In order to create the root directory, you need to do the following steps:

- Mark the `ROOT_INUM` inode as used in the inode bitmap. This means you need to set bit `ROOT_INUM (1)` to a value of 1
- Find an unused data block by calling `find_free_block()`, and also mark it as used in the data block bitmap
- Add two `mem_dirent` entries in the data block that we found in the previous step. The first one must be called `.` (dot) and represents the current directory, so its `inum` is `ROOT_INUM`, and the second one is called `..` (two dots) and represents the parent directory. Since the root has no parent, this entry's `inum` should also be set to `ROOT_INUM`
- Create a new inode at position `ROOT_INUM` in the inode blocks and initialize it with the necessary information such as the size in bytes used in the data block for the two entries in the last step, or the current time (hint: use the `time()` function in `time.h`). The `num_links` can be set to 1 initially.

### 5. `inode_ref find_inode_from_path(void *fs, char *path)`

Given a string with the form `"/dir1/dir2/file"`, return the inode number of the last part of the path (in this case `"file"`). These are the steps to perform this:

- Start at the root inode, which is located at `ROOT_INUM`, retrieve the inode struct
- Break the `path` string at the slash character position (in the previous path, you would get `"dir1"`, `"dir2"`, `"file"`) (Hint, use the `strtok()` function from `string.h` to do this)
- For each the the entries in the previous step, do the following:
  1. Find the entry in the current inode data blocks as a `mem_dirent` (for example, in the previous case you'd start by finding an entry called `"dir1"` in the root inode contents)
  2. Get the inode number from the `mem_dirent` entry, and locate the new inode struct from this number, and repeat step 1 for the next part of the path until you are done
  3. Return the last inode number you found

### 6. `inode_ref create_dir(void *fs, char *parent_path, char *dir_name)`

This function calls `find_inode_from_path()` to get the inode of the `parent_path`, then it creates a new `mem_dirent` in that inode for a new directory called `dir_name`. This is the same process as in `create_root_dir` but now we'll need to allocate a new inode and block for `"dir_name"`. For this function, you'll need to add the new entry to the parent inode, and create the `.` and `..` entries for `dir_name`.

Extra credit

### 7. `inode_ref create_file(void *fs, char *parent_path, char *file_name, char *contents)`

Same as `create_dir()`, but it creates a regular file with the contents in `contents` (which is a NULL-terminated string).

### 8. Deal with indirect block pointers

You will get full credit for finishing functions 1 to 6 without handling the case for indirect block pointers (i.e., inodes will just use blocks that contain actual data). To get additional extra credit, implement function 7 so that it can store data in additional blocks referenced by indirect pointers (that is, after your inode uses all direct pointers, it will use the indirect pointer block to store additional block references to other blocks with data)

## Notes

- We will see all the functions during the labs, so make sure you attend the next labs or watch the videos if you have questions about this assignment. Also, make sure you review Chapter 40 of the textbook for a more in-depth explanation of all the concepts.
- Don't worry about the `num_links` variable in inodes. For the purposes of this assignment you can set it to 1 when you create a new file.
- Have in mind that the whole filesystem will be in memory. You will allocate all of it in `makefs()`. You will also have to write some helper functions to convert from inode numbers to `struct mem_inode` pointers that reference the right position in memory, and also from data block numbers to `void *` pointers to the actual blocks.
- Upload your `memfs.c` file with your function implementations for the submission.

## Files

[memfs.h](https://fiu.instructure.com/courses/109531/files/19483886/download?download_frd=1)  ([https://fiu.instructure.com/courses/109531/files/19483886/download?download\\_frd=1](https://fiu.instructure.com/courses/109531/files/19483886/download?download_frd=1))