552_lfs

To compile, run

```
make mklfs
make lfs
```

Note the command `make` will compile **only** lfs, and will not compile mklfs.

Then, initiate an empty disk with

```
./mklfs -b BLOCK_SIZE -l SEGMENT_SIZE -w WEAR_LIMIT FILE_NAME
```

And start up the file system with

```
./lfs FILE_NAME MNT_PNT
```

Summary

What works, what doesn't work?

All required fuse operations work in memory. Once a segment has been fully assigned blocks, it is written to disk (can be checked with hexdump). The lfs does not correctly load the filesystem on boot; It always treats it like an empty disk. Checkpoints are only half-implemented.

We have all of the required fuse operations working in memory;

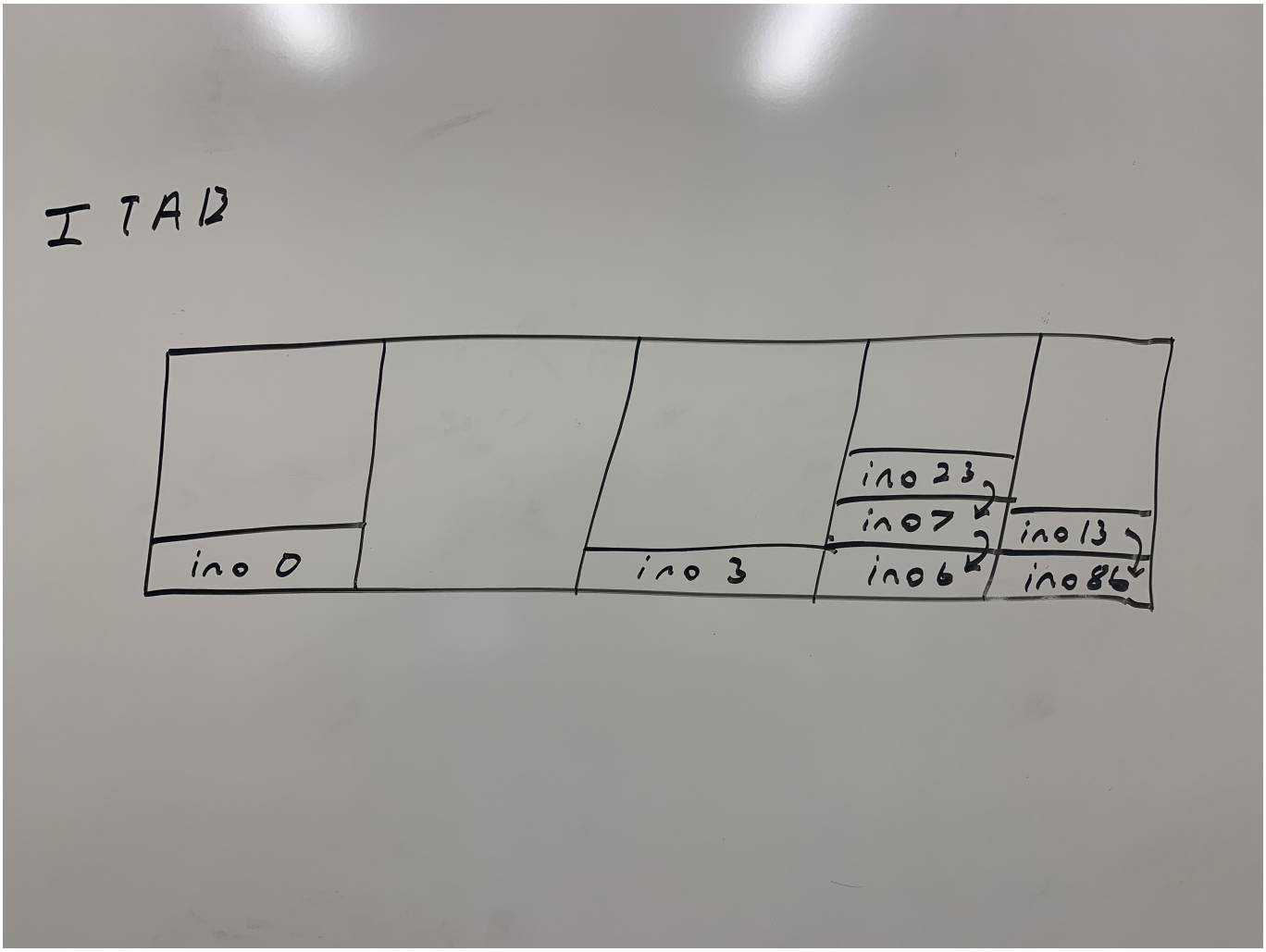
- readdir: First looks up the inode associated with the path formal parameter and ensures that it is a DIR_TYPE. We have implemented inodes as a tree (linked list) so we iterate over all children of the directory and pass it to the filler() function given by fuse.
- getattr: Sets the appropriate info in the stat struct passed by fuse. Currently gives uid, gid, atime, mtime to all inode types. The mode is set based on file type; S_IFDIR | 0755 if directory and S_IFREG | 0644 if file; Links are yet to be implemented. Additionally number of links are hard set to 1 since no links yet.
- utimens: returns 0.
- create: First creates a new inode with the name path passed to it by fuse, and is inserted into the inode hashtable. Then, the inode is appended to it's parent directory (hard coded to the root dir for now). The file is initialized empty and size set to 0.

- open: Our current implementation does not need open for functionality; returns 0.
- release: Our current implementation does not need release for functionality; returns 0.
- read: Looks up the inode associated with path and uses memcpy to load it into the fuse buf. For now, we only handle files that are <= one block in length; If bigger, we return an error.
- write: Looks up the inode associated with path and uses memcpy to write it to the block_addr data structure. Currently only handles files of at most 1 block in length. If bigger, we return an error.
- truncate: Needed for write. Looks up the inode associated with path and sets it's size to the passed paramater.

---

inode table

We implemented inodes perhaps a bit strangely. We don't know ahead of time how many inodes we might need, so an array can be an inefficient data structure to store them despite it's ease of use; i.e. we might have to resize the array many times. We elected to use a hash table instead. The implementation we settled on was a table of fixed sized (NUM_BUCKETS = 256), and each inode is assigned a bucket based on a hash of it's name. Each bucket contains a linked list of inodes, so lookup can be O(n) in the worst case but shouldn't happen if the hash function is pseudo uniform. This was motivated by the fact that fuse only gives us a file via it's path, so lookup via path seemed the easiest to interface with. The inode hashtable is a static global ITAB available in inode-tab.c; An image abstraction is shown below.

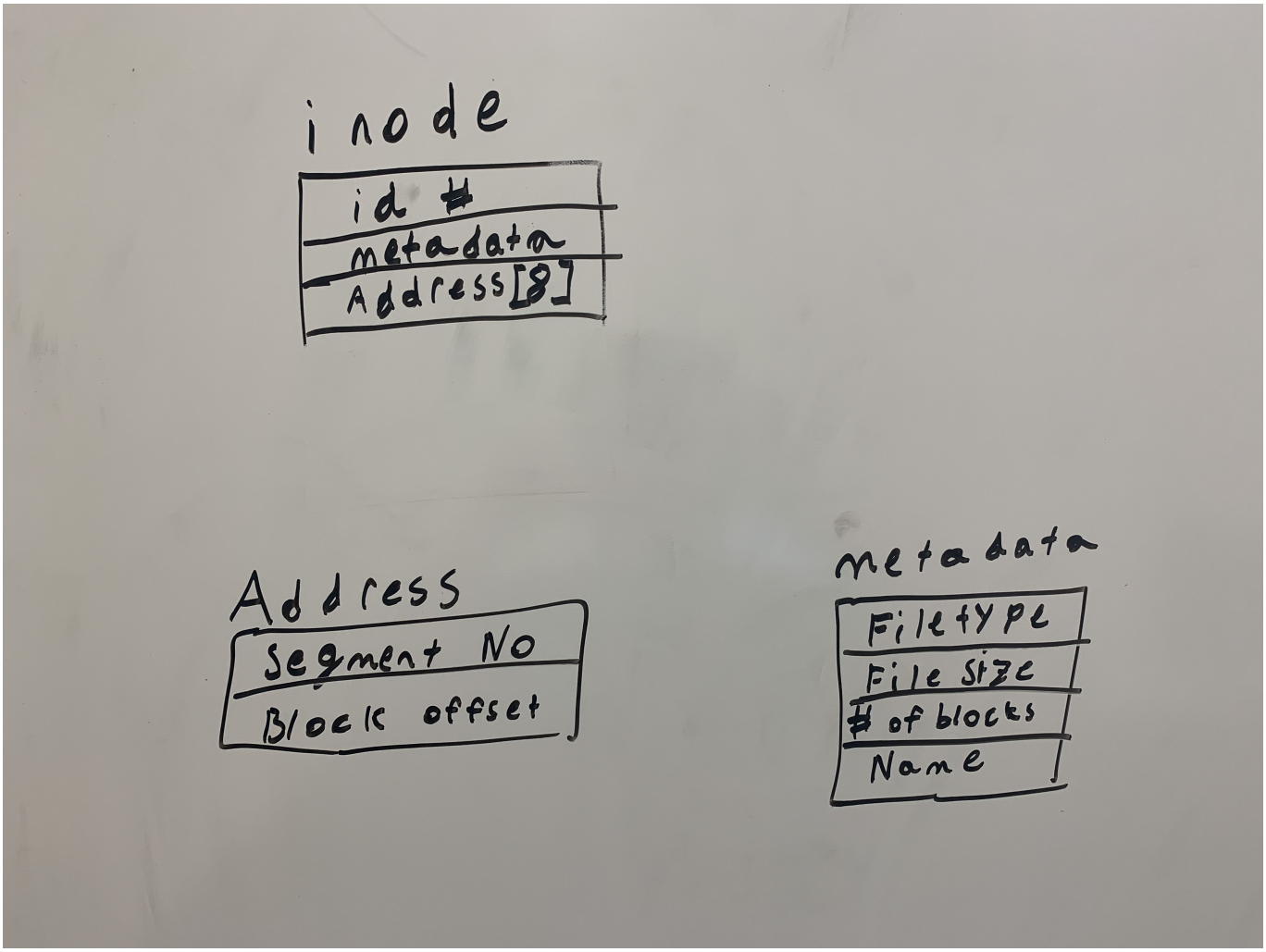


inodes

The inodes themselves are implemented as simple structs with 4 fields: the inode number, a metadata struct, an array of address structs, and a pointer to the next inode in the list (possibly null).

metadata: The metadata struct (simply named meta) contains various fields that might be good to have on hand; We have given it a file type (enum with FILE_TYPE, DIR_TYPE, and LINK_TYPE fields), a size in bytes, the number of blocks the file uses, and it's name as a string.

address: The address struct (named block_addr) contains the segment number and block number the file data can be found on disk. This is also where we store the buffer for the data in memory, in the field buf which is allocated BLOCK_SIZE bytes when the block_addr struct is initialized. As mentioned, an inode contains an array (hard coded to size 8) of block_addresses, so the max file size we could support is 8 blocks. We do not implement indirect addressing.



Logging

Currently, the `log' portion of our log based file system is not quite correct. While we do greedily assign new blocks to the end of the current segment and always write out new blocks (no modifying old ones). The only thing really missing is a fully functional checkpoint system, which we plan to implement by making note of all of the segment tables inode structs and writing them to disk at segment 0.

Challenges

We found it quite difficult to decide on implementation details. While we understood LFS at a very high, conceptual level it felt impossible to know where to even start. The fuse callbacks were concrete, already implemented functions so it felt easiest to start by designing around those. This may have been a little backwards (e.g. designing the inode table as a hashmap, since that feels natural when given a string).

C is not the language of choice for either of us, so many many bugs and segfaults were battled against while writing this code. There is a known memory leak in inode-tab.c, in flush_to_log(), currently line 80. Sometimes this points to unallocated memory. While we have not observed this yet calling a crash, it likely would if the fs is run long enough. This unfamiliarity made development time take longer than either of us would like for a class project.

Currently however, there should be no other seg faults or errors as long as files aren't too big. Many fuse operations that aren't detailed above are unsupported. There are likely more memory leaks however. We aimed to make the code concise but readable.

Files

mklfs.c

mklfs.c creates a flash memory device by taking input arguments from the command line, creating a file with the specified name and size, and writing a header to the file containing disk parameters. The fill_device function initializes the device header data and writes it to the file using Flash_Write function from the "flash.h" library.

log.h

log.h is used to represent information about a segment.

log.c

log.c writes the buffer of an i_node structure to a specific sector of a flash file and returns 0 if the write was successful.

lfs.h

lfs.h declares and defines various functions for FUSE callbacks and other helper functions for loading a Flash device, initializing the root directory, and setting file metadata.

lfs.c

lfs.c is the implementation of the a FUSE-based filesystem in C, with a specific set of operations defined in the "struct fuse_operations".

dump_flash.c

dump_flash.c reads data from a flash memory file mentioned in the form of command line argument, by opening the file using Flash_Open function from "flash.h" library and checking for any errors.

dir.c

dir.c appends a file inode to a directory inode by traversing the linked list of inodes until the end is reached and adding the new inode as the last element in the list.

global.h

global.h contains includes for libraries common to all (most) .c files

types.h

types.h contains the various structs, typedefs, and datastructure defs needed for LFS

inode-tab.c

inode-tab.c is responsible for managing the inode table.

test.py

Python script we used to systematically test our file system. Includes trying commands like touch, write, cat, cp, sha256, and diff.