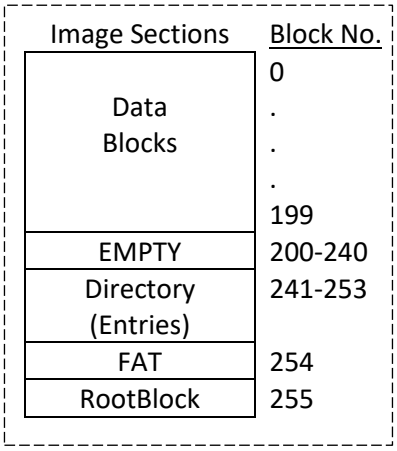
Implementation

From libfuse documentation:

“libfuse offers two APIs: a "high-level", synchronous API, and a "low-level" asynchronous API. In both cases, incoming requests from the kernel are passed to the main program using callbacks. When using the high-level API, the callbacks may work with file names and paths instead of inodes, and processing of a request finishes when the callback function returns. When using the low-level API, the callbacks must work with inodes and responses must be sent explicitly using a separate set of API functions.”

The original academic project described a 128kb image, composed of 256, 512-byte blocks. 200 devoted to user data, 41 empty, 13 for the directory structure, 1 for the FAT, and 1 root block at the end.

High-level API took care of path-to-inode mapping, and various other operations, but only for the developer who could successfully traverse the quagmire of FUSE documentation. Until very recently, I was not such a developer. Consequently, MOOFS is implemented using the low-level API, trading large scale performance and abstraction for absolute control over every aspect of the design. The amount of boilerplate may have increased dramatically, but at the very least it was boilerplate with which I was intimately familiar.



libfuman

MOOFS uses *libfuman*, a custom backend, to handle user data. With FUSE documentation so difficult to understand, handling user data entirely within the MOOFS client was a nonstarter. In order to guard against later fallout from flawed FUSE assumptions, a custom backend was designed to logically handle filesystem operations, independent of libfuse. Critical metadata structures are fileObjects and fileBlocks, managed by a fileUnitManager.

```
//fileObject
//"represents a file"
/*
*   type     - not currently in use
*   active   - 1(active, not queued)/0(inactive, queued)
*   len      - number of bytes managed by fileObject
*   fObj_id  - id of fileObject
*   dfe      - struct directoryFileEntry for fileObject
*   HT_PN    - neighbors in queue OR head/tail of fileBlocks
*/
struct fileObject{
    char type;
    char active;
    unsigned int len;
    int fObj_id;
    dFEnt* dfe;
    void* HT_PN[2];
};
```

An active fileObject(fObj) SHALL maintain all information necessary to support read and write operations; however, it DOES NOT have authority to manipulate fileBlocks. When inactive, it is part of a queue controlled by the fUMan. To save space, a fileObject maintains HT_PN, a multipurpose array of void pointers.

Suppose initially, a fObj is inactive. Its HT_PN contains fObj* prev and fObj* next, its neighbors within the inactive fObj queue. Inactive fObjs are to be distributed by the fileUnitManager as needed. When a new file is created, the fUMan dequeues a fObj, toggling it active and passing a directoryFileEntry pointer. The newly activated fObj's HT_PN now contains fBlk* head and fBlk* tail, which impose bounds on the accessible data. fBlk* tail can be moved to either extend a file(adding blocks) or reduce it(removing blocks).

The fUMan may see fit to reduce a fObj, removing *K* of its file blocks. In this case, the fObj's tail, HT_PN[1], is moved backwards along the double linked list *K* fBlks,(via pointers), assigning a new tail to the fObj. Then, the fUMan assigns the dump node's HT_PN[0]=new tail->next, HT_PN[1]=old tail, as if to creating a new file of size *K* and deactivates the node. These now-deactivated fBlks are added to queue of inactive fBlks, to be distributed by the fUMan as needed. Deleting an fObj is reduction to 0, at which point the fObj is added to the queue of inactive fObjs by the fUMan.

```
//fileBlock
//"makes up part of a file"
/*
*   type     - not currently in use
*   active   - 1(active, not queued)/0(inactive, queued)
*   fObj_id  - id of owner fileObject, -1 if free
*   fBlk_id  - id of fileBlock
*   data     - pointer to fileBlock data,SZ_BLOCK byte buffer
*   prev     - addr of prev fileBlock
*   next     - address of next fileBlock
*/
struct fileBlock{
    char type;
    char active;
    int fObj_id;
    int fBlk_id;
    unsigned char* data;
    fBlk** prev;
    fBlk** next;
};
```

An active fileBlock(fBlk) holds a series of bytes belonging to a fileObject. prev and next point to the previous and next fBlks belonging to the same fObj_id. When fBlk->Prev==NULL, this is the start Block, and when fBlk->next==NULL, it is the end Block. Otherwise, prev and next may contain a series' of bytes. When inactive, a fBlk is part of a queue maintained by the fUMan, to be distributed to fileObjects when necessary.

The fBlk SHALL keep track of its neighbors, whether it is part of a fObj's chainor the inactive queue. Extend, reduce, and data buffer management are delegated to the fUMan.

fileUnitManager

A fileUnitManager is the highest level of abstraction within *libfuman*, capable of interacting directly with the MOOFS client.

```
//fileUnitManager
//"controls the entire image"
/*
*   imgName     - name of src image file
*   rb          - struct rootBlock for image file
*
*   avail_fObjs - (queue len)fileObjects available for assignment
*   max_fObjs   - maximum number of fileObjects initialized
*   arr_fObjs   - array of fileObject pointers
*   fObjQ_head  - head of fileObjectQueue
*   fObjQ_tail  - tail of fileObjectQueue
*
*   avail_fBlks - (queue len),fileBlocks available for assignment
*   max_fBlks   - number of fileBlocks initialized
*   user_fBlks  - max num of fileBlocks available for asignment
*   arr_fBlks   - array of fileBlock pointers
*   fBlkQ_head  - head of fileBlockQueue
*   fBlkQ_tail  - tail of fileBlockQueue
*/
struct fileUnitManager{
    char* imgName;
    rBlk* rb;

    int avail_fObjs;
    int max_fObjs;
    fObj** arr_fObjs;
    fObj* fObjQ_head;
    fObj* fObjQ_tail;

    int avail_fBlks;
    int max_fBlks;
    int user_fBlks;
    fBlk** arr_fBlks;
    fBlk* fBlkQ_head;
    fBlk* fBlkQ_tail;
};
```