Design Document for Assignment 4 - FAT File System

Kenneth Mayer
Marian Vladoi

March 18, 2018

Contents

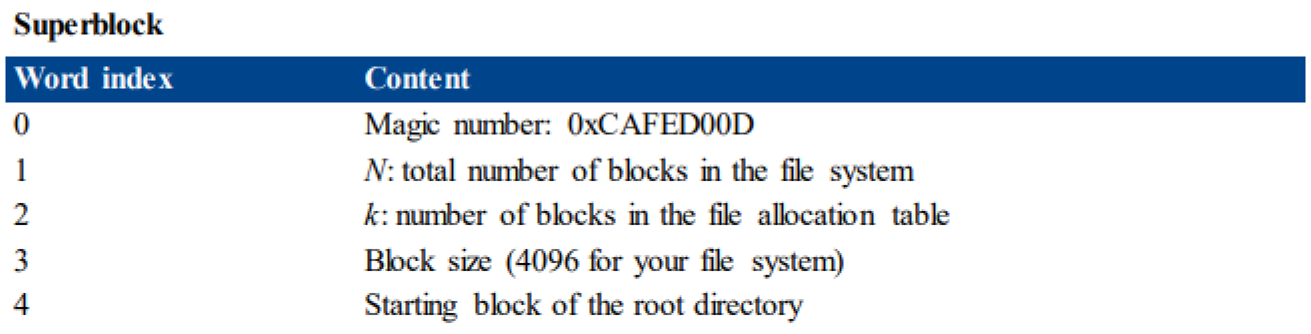
1 Introduction

1.1 File System

In this assignment, we were to use the FUSE tool (**F**ile System in **Use**r Space) to implement a FAT32 file system with a block size of 4KB. A file system is a specified format for storing and accessing data on a disk; because everything is a file in UNIX, we are able to use a file to act as our disk.

The FAT (File Allocation Table) file system is one of the simplest file systems, and is suitable for small disk and file sizes, though it scales badly. The core of the FAT system is the titular file allocation table, which is simply a linked list of, in this case 32-bit integers (hence the FAT32 moniker), that correspond to blocks on disk where a particular file's data is stored. A particular entry in the FAT points to the next block in that file, or to an EOF symbol if there is no additional data. The FAT also contains entries for the FAT itself, which are given a special symbol to indicate that that particular block should never be reallocated.

To identify our file system, we use a superblock placed at the beginning of the disk, which has the following format:

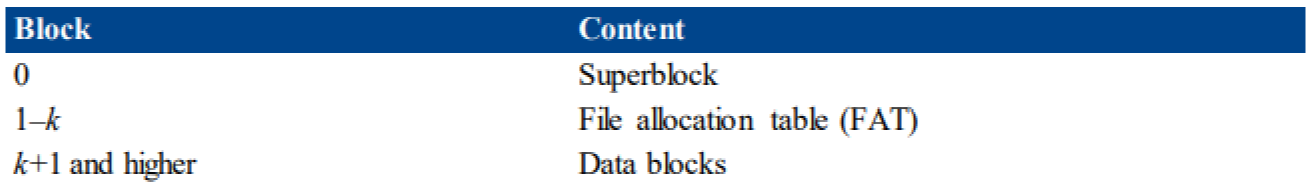


Superblock

Word index	Content
0	Magic number: 0xCAFED00D
1	*N*: total number of blocks in the file system
2	*k*: number of blocks in the file allocation table
3	Block size (4096 for your file system)
4	Starting block of the root directory

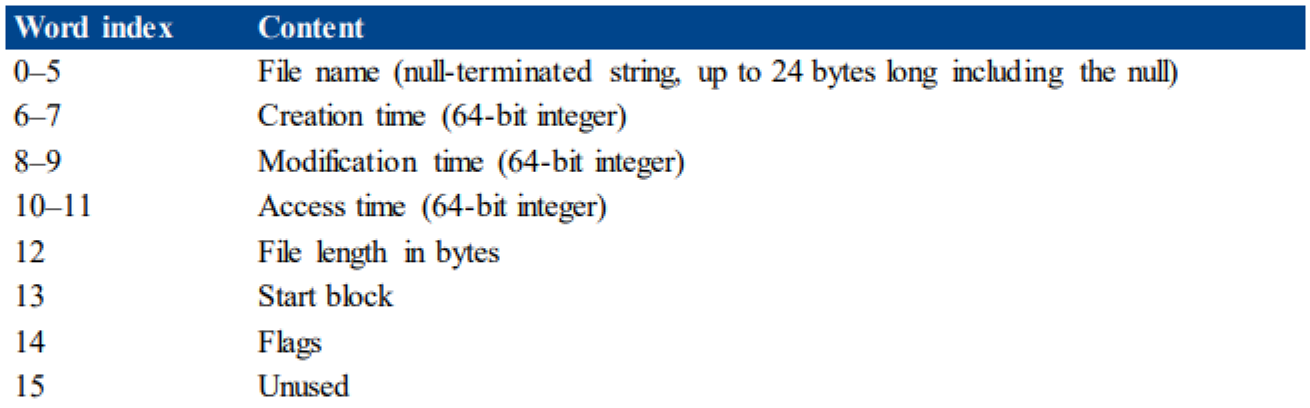
The magic number can be checked to ensure that the file system has been formatted correctly.

The overall format of the disk (and therefore the FAT) looks like:



Block	Content
0	Superblock
1–*k*	File allocation table (FAT)
k+1 and higher	Data blocks

Each directory entry has the following format:



Word index	Content
0–5	File name (null-terminated string, up to 24 bytes long including the null)
6–7	Creation time (64-bit integer)
8–9	Modification time (64-bit integer)
10–11	Access time (64-bit integer)
12	File length in bytes
13	Start block
14	Flags
15	Unused

Because the directory entry has a pointer the starting block of the file data, the FAT will not have a linked list in it unless there are files larger than one block.

1.2 FUSE

FUSE is a tool that allows users to create their own file system without modifying the kernel; its functionality is controlled by implementing a set of callback functions which FUSE will execute depending on which commands it receives.

Interestingly, FUSE does not really care about your disk file, it can be used, but all of the logic that reads and performs operations on it is specific to your implementation. All that matters is that you return what the callback functions expect to be returned.

2 Submitted Files

```
fat_fuse.c  // The main file, contains all of our fuse routines and helper functions
init_disk.c // A file that creates a properly formatted FAT disk of arbitrary size
Makefile    // Contains targets for compiling the previous
```

3 Compilation Instructions

For this assignment, we used FUSE 3. At the time we came to this decision, we weren't given any explicit instructions regarding which FUSE version to use, and we didn't want to rework the program for normal use.

On our systems, before compiling, we had to enter

```
export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
```

Before the file would compile correctly, so that may have to be done during testing as well.

After that is completed, `make`, will compile and the command

```
./fat_fuse [mount point] [disk file]
```

Will mount the disk file in directory mount point.

If you want to test the program that initializes the new disk, `make init_disk` and

```
./init_disk [size in blocks] [disk file]
```

Will create a new file of the proper size in disk file.

4 Formatting

To check for uninitialized data, I place the magic number 0xF5EC in the unused word. This way, it is very unlikely that garbage data will be interpreted as valid.

5 Testing Methodology

To test the drive, mainly our methods were to mount the drive and try to perform the expected operations on it. For example, after writing our `mkdir` routine, we would try to create a directory in our mounted FS.

Basically, we just did operations on the disk many times over tried to find edge cases (for example, our `mv` routine would work unless the target was the root directory, because we hadn't written a special case for root). To test reading and writing, we would use `emacs` to create multi-block files, save them, and then see if they were correctly read. Additionally, we would look at the disk file to see if we could spot the different data stored on it.

We would then mount the modified disk a new directory to make sure that the disk had correctly retained the data and could be modified again. This revealed some issues, like how we had forgotten to update the file size after writing to it and how we had been accidentally writing over the superblock in our `free_file ()` routine.

6 Structs

There are a few structs that we use in our program to make it easier to access the attributes we're looking for. These are listed below

6.1 Superblock

We access the superblock occasionally, but most of these members were used mainly for our initialization program

```
struct superblock{
  int32_t magic_number;   /* The magic number identifies the FS */
  int32_t N;              /* The total number of blocks in the system */
  int32_t k;              /* Number of blocks in the FAT */
  int32_t block_size;     /* Size of the block in the file system */
  int32_t starting_block; /* Block number for the starting block (k+1) */
}
```

6.2 Directory Entry

This is a struct that we use a lot; almost in every function. It shows the structure of a directory entry.

```
struct directory_entry{
  char   file_name[NAME_LENGTH]; /* File name, up to 24 character */
  uint64_t create_time;          /* The file's creation time */
  uint64_t mod_time;             /* The file's last modification time */
  uint64_t access_time;          /* The file's last access time */
  int32_t file_length;           /* The files's length in bytes */
  int32_t start_block;           /* Starting block number of the file */
  int32_t flags;                 /* Two flags; 0=file, 1=directory */
  int32_t unused;
}
```

6.3 FUSE Context Private Data

In FUSE, you can pass a pointer to the fuse_main call, which can later be accessed by all fuse functions, sort of like global variables, but that won't interfere with other contexts.

We passed the following struct:

```
struct fuse_info{
    char *file_name;    /* contains the path name, currently unused */
    int fat_fd;         /* file descriptor for the disk file */
    FILE *fuse_log;     /* file stream for debugging output */
    superblock *super;  /* file system superblock */
    int32_t *fat_table; /* File system FAT*/
}
```

7 Helper Functions

Many operations on your file system will be performed over and over; for example, in this file system, in almost every callback function, I had to find the directory entry corresponding to the path the operation was being performed on.

7.1 int32_t find_path_block(int32_t fuse_info *my_fuse, char *path, int32_t *dirent_index)

This is one of the most important helper functions. It accepts a path, for example `/foo/bar/file.txt` and returns, the block containing the path's directory entry, and it also returns in `dirent_index`, the index of the directory entry that contains the path's directory entry.

It operates as follows:

```
tokenize path along the "/" character
get the starting block of the root directory

loop over the number of directory levels in the path (2 iterations in the example)
    block = search_directory(start, current path token, dirent_index)
    if(not found)
        return -1
    read block
    start = block[dirent_index]
```

```
return block
```

7.2 int32_t search_directory(int32_t fuse_info *my_fuse, int32_t num, int32_t *dirent_index)

This function accepts the fuse context private data and a block pointing to a directory and returns the block that contains the directory entry on disk and the index of that dirent if it can find it, -1 if it cannot.

```
Read the starting block of the directory
loop over all the block's directory entries
    if(found)
        return offset
    if(another block in the FAT)
        return recursive call on next block
return −1
```

7.3 int32_t find_free_dirent_block(fuse_info *my_fuse, int32_t num)

This function accepts the fuse context private data and a block pointing to the location of a directory block on disk. It searches this block until it finds a free directory entry, then returns the block containing that free entry and the index of the free entry in the block.

```
Read the starting block of the directory
loop over all the blocks directory entries
    if(dirent's unused member is not our magic number)
        return offset
if(current directory has another block)
    return recursive call on next block
else
    find the next free entry in the table (call find_free())
    assign that entry in the fat table to EOF
    recursive call on new block
return −1
```

7.4 int32_t free_file(fuse_info *my_fuse, int32_t start_block)

This function accepts the fuse context private data and a the starting block of the directory entry to freed. It frees a FAT entry and also has to overwrite that file's data with zeroes due to some of our design choices.

```
overwrite the first block that was passed with zeroes
while(next block != EOF)
    retrieve next block
    assign current block to next block
    free next block
    overwrite this block with zeroes
free the last block with the EOF in it
return 0
```

The implementation of this code is a bit strange and should probably be revised.

7.5 int32_t is_dir_empty(fuse_info *my_fuse, int32_t num)

This function accepts the fuse context private data and the starting block of a directory. It checks if the directory is empty.

```
Read the starting block of the directory
ret = 0
loop over all the blocks directory entries
    if(dirent's start block is equal to 0)
        if(directory is not name ".." or ".")
            return -1;
if(current directory has another block)
    return recursive call on next block
return 0
```

7.6 int32_t write_file(fuse_info *my_fuse, const char *buffer, int32_t num, off_t offset, size_t size)

This function is called by the FUSE write callback function. It accepts our fuse context private data, a buffer containing the data to write, an offset specifying where to start writing and the number of bytes to write.

This implementation of this function is complicated by the possibility that both the offset and the amount of data to write are larger than one block.

```
seek_blocks = floor(offset/block_size)
for 0 to seek_blocks-1
    follow the linked list
/* How much offset is left once we reach this block? */
relative_offset = offset - block_size * seek_blocks
seek to num * block_size + relative_offset

/* Do we need to move to blocks beyond this one? */
if(size > block_size-relative_offset)
    ret += write block_size - relative_offset bytes
    size -= block_size - relative_offset
else
    ret = write size bytes
    return ret
write_blocks = size/block_size

for 0 to write_blocks-1
    next_num = fat_table[num]
    if(next_num is EOF)
        find the next free block (call find_free())
        assign the entry to that block as EOF
        num = fat_table[num] /* now the next free block */
    else
        num = next_num
    seek to the start of block num
    ret =+ write block_size bytes
    size -= block_size

/* basically one final iteration with one change */
next_num = fat_table[num]
if(next_num is EOF)
    find the next free block (call find_free())
    assign the entry to that block as EOF
    num = fat_table[num]
else
    num = next_num
seek to the start of block num
```

```
ret =+ write size bytes

return ret
```

7.7 int32_t read_file(fuse_info *my_fuse, const char *buffer, int32_t num, off_t offset, size_t size)

This function is very similar to write_file with one key difference - it can't create new FAT blocks.

```
seek_blocks = floor(offset/block_size)
for 0 to seek_blocks-1
    follow the linked list
/* How much offset is left once we reach this block? */
relative_offset = offset - block_size * seek_blocks
seek to num * block_size + relative_offset

/* Do we need to move to blocks beyond this one? */
if(size > block_size-relative_offset)
    ret += read block_size - relative_offset bytes
    size -= block_size - relative_offset
else
    ret = write size bytes
    return ret
write_blocks = size/block_size

for 0 to write_blocks-1
    num = fat_table[num]
    seek to the start of block num
    ret =+ read block_size bytes
    size -= block_size

/* basically one final iteration with one change */
next_num = fat_table[num]
seek to the start of block num
ret =+ read size bytes

return ret
```

8 Callback Functions

These are the functions that FUSE will call based on the command it receives. It's up to the user to choose which functions to implement. Here is a list of the ones that we have chosen to implement

```
static struct fuse_operations fat_operations = {
  .init     = fat_init,
  .destroy  = fat_destroy,
  .mkdir    = fat_mkdir,
  .getattr  = fat_getattr,
  .readdir  = fat_readdir,
  .read     = fat_read,
  .unlink   = fat_unlink,
  .write    = fat_write,
  .rmdir    = fat_rmdir,
  .rename   = fat_rename,
  .open     = fat_open,
  .create   = fat_create,
  .access   = fat_access,
  .truncate = fat_truncate,
};
```

Now let's go into some detail about how each one works.

8.1 static void *fat_init(struct fuse_conn_info *conn, struct fuse_config *cfg)

This function simply returns the struct we passed to `fuse_main()`.

```
retrieve private data
return private data
```

8.2 void fat_destroy(void *private_data)

I'm not entirely sure what this function is supposed to do or when it's called, but we don't really have any functionality implemented here.

8.3 static int fat_getattr(const char *path, struct stat *st, struct fuse_file_info *fi)

This function returns the attributes of file located at path.

```
get private data
store st_uid, st_gid, st_dev, st_ino and st_blksize into st
allocate a directory entry
if(path is "/" (root))
    (there is no dirent for root, so these are sort of 'dummy' values)
    (with some work, this could be more accurate, but I don't think it matters much)
    store st_blocks = 1 into st
    store st_size = 64 into st
    store st_mode = rwx and is_directory into st
    store st_nlink = 2 into st
else
    find path's directory entry (call find_path_block())
    read dirent
    if(directory not found)
        return -ENOENT
    store st_blocks = ceiling(dirent.file_length/block_size) into st
    store st_size = dirent.file_length into st
    store st_nlink = 1 into st
    store st_atime = dirent.access_time into st
    store st_mtime = dirent.mod_time into st
    if(file is a directory)
        store st_mode = rwx and is_directory into st
     else
        store st_mode = rwx and is_file into st
return 0
```

8.4 static int fat_readdir(const char *path, void *buffer, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi, enum fuse_readdir_flags flags)

This file reads the contents of a directory into buffer.

```
get private data
allocate a directory entry
if(path is root)
    num = super.starting_block
else
    find path's directory entry (call find_path_block())
    if(directory not found)
        return -ENOENT
    read dirent
    num = directory.start_block

while(num != FAT_EOF)
    read block num
    iterate over read block's directory entries
        if(current directory's start block isn't 0)
            filler(buffer, directory name)
    num = next block in the linked list

return 0
```

8.5 static int fat_read(const char *path, char *buffer, size_t size, off_t offset, struct fuse_file_info *fi)

This function reads size bytes from the disk starting at offset bytes.

```
get private data
allocate a directory entry
find path's directory entry (call find_path_block())
if(directory not found)
    return -ENOENT
read dirent
ret = call read_file()
return ret
```

8.6 static int fat_write(const char *path, const char *buffer, size_t size, off_t offset, struct fuse_file_info *fi)

This function writes size bytes to the disk starting at offset bytes.

```
get private data
allocate a directory entry
find path's directory entry (call find_path_block())
if(directory not found)
    return -ENOENT
read dirent
ret = call write_file()
return ret
```

8.7 static int fat_rmdir(const char *path)

This function removes a directory as long as it is empty.

```
get private data
allocate a directory entry
find path's directory entry (call find_path_block())
```

```
if(directory not found)
    return -ENOENT
read dirent
check if directory is empty (call is_dir_empty())
if(not empty)
    return -ENOTEMPTY
free the directory (call free_file())
write zeroes over path's dirent
return 0
```

8.8 static int fat_rename(const char *path, const char *name, unsigned int flags)

This functions renames path to name. It is one of the more complicated ones; there are three cases that must be considered during a call

1. path and name are in the same directory - it's easy, all you have to do is change the dirent's file name
2. path and name are in a different directory - this means you need to erase the directory entry at path and move it to name
3. path and name are in a different directory and a file with name already exists at the target path - this means you have to move path to name and erase the file that already exists at name

```
get private data
allocate a directory entry
check whether name already exists (call find_path_block())
if(name already exists)
    read name's dirent
    free name's data (call free_file())
    write zeroes over name's dirent
retrieve the actual file names (characters after the last slash in name and path)
check if the target directories (everything before the last slash) are the same
if(target directories are the same)
    move = 0
find the directory entry for path (call find_path_block())
if(directory not found)
    return -ENOENT
read path's dirent
if(move = 0)
    write new file name to path's dirent.file_name
    overwrite directory entry with new one
    return 0
else
    overwrite path's dirent with zeroes
    write new file to path's dirent.file_name
    find the directory entry for the directory the new file will be placed in
    read said directory's starting block
    find a free dirent (call find_free_dirent())
    if(no free dirent)
        return -1 note: this means the disk is full
    write new directory entry
    return 0
return -1 (hopefully unreachable)
```

8.9 static int fat_open(const char *path, struct fuse_file_info *fi)

Because we aren't keeping track of permissions, this function just asserts that a file exists.

```
get private data
allocate a directory entry
find path's directory entry (call find_path_block())
if(directory not found)
    return -ENOENT
return 0
```

8.10 static int fat_create(const char *path, mode_t mode, struct fuse_file_info *fi)

This functions creates a new file. This means it must create a directory entry and find a starting block for it.

```
get private data
allocate a directory entry (directory)
allocate a directory entry for our new file (new_dir)

assign attributes to new_dir
  new_dir.create_time   = current time
  new_dir.mod_time      = current time
  new_dir.access_time   = current time
  new_dir.file_length   = 0
  new_dir.flags         = 0
  new_dir.unused        = 0

find the actual file name (everything after the last slash)
copy this to new_dir.file_name

if(target directory is root)
    find a free dirent in root
else
    find the target directory's dirent (call find_path_block)
    read target directory's dirent

new_dir.start_block = find_free()
write EOF to the found FAT entry
write new_dir

return 0
```

8.11 static int fat_mkdir(const char *path, mode_t mode)

This function creates a new directory if one with the same name doesn't already exist.

```
get private data
allocate a directory entry (directory)
allocate a directory entry for our new directory (new_dir)

assign attributes to new_dir
  new_dir.create_time   = current time
  new_dir.mod_time      = current time
  new_dir.access_time   = current time
  new_dir.file_length   = 0
  new_dir.flags         = 1
  new_dir.unused        = 0

remove the final slash so we can find the directory our new directory will be placed in
save the directory name
```

```
if(target directory is root)
    root = 1 (we'll need this later)
    find a free dirent in root (call find_free_dirent())
else
    root = 0
    find the target directory's path (call find_path_block)
    if(directory not found)
        return -ENOENT
    find a free dirent in the target directory (call find_free_dirent())
new_dir.start_block = find_free()
assign EOF to the found FAT entry\
seek to the new starting block

allocate two new dirents (for "." and "..")
assign attributes to .'s dirent as with new_dir
assign .'s dirent.start_block = new_dir.start_block
assign attributes to ..'s dirent as with new_dir
if(root = 0)
    assign ..'s dirent.start_block = target dirent's.start_block
else
    assign ..'s dirent.start_block = super.starting_block

return 0
```

8.12 static int fat_unlink(const char *path)

In our file system, this function deletes a file.

```
get private data
allocate a directory entry

find directory entry of path (call find_path_block())
read path's dirent
free_file(path's dirent.start_block)
overwrite path's directory entry with zeroes

return 0
```

8.13 static int fat_access(const char* path, int mask)

Identical to `fat_open()`.

9 Known Issues

There are still a few issues plaguing our implementation of the file system, though most of the functionality is fine.

1. Memory leaks - A lot of my functions leak memory (calloc'ed data is not properly freed). This could be fixed relatively easily, but it might take a while. I don't want to risk breaking the program at this point.
2. Touching a file twice - when you use `touch` on a file that already exists, FUSE returns a "function not implemented" error.

3. We don’t know what happens when the disk runs out of space
4. “.” entry lists modification time as 0 in the root, at least on the disks we made