CMPS 111 - 01 (Darrell Long)
Assignment 4
Winter 2017

CMPS 111 - 01 Fats Domino File System

Erik Andersen (Team Captain), Michael Cardoza, Yuzhuang Chen, Seongwoo Choi

Project Objective:
In assignment 4, we are incorporating the FUSE file system framework to implement a simple FATS Domino Filesystem in userspace. This assignment is different from the last two assignments since the code of file system runs in userspace even though it is closely related to the operating system kernel. This assignment will give an experience to extend what is in the operating system and user code. This will help to implement new file system functionality on the FreeBSD.

Project Specifies:
A file system is a part of the FreeBSD operating system that deals with the file management on a disk. There are many different ways of integrating file system in structure. For instance, one can use linked lists or bitmaps to integrate with the file system. An access time and efficient disk utilization may be different from each other depending on the structure that has been chosen. Because of this reason, in this assignment, our team is implementing a FAT (File Allocation Table) structure to manage files on a disk and also, this structure is similar to a linkedlist file system structure, but we need to use a FAT system to trace back the record of contiguous disk blocks. The file system structure is simple to implement. Moreover, it has an advantage and a disadvantage. The advantage of using a FAT structure instead of a linked list is that FAT allows a faster random memory access and reduces fragmentations within each disk block in a memory. Meanwhile, the disadvantage is that it requires the table to be stored in memory.

Project Overview:
In order to implement the File Allocation Table file system on a disk, we need to have a disk and a framework to implement our file system routines. In addition to that, a user space program would be needed, so that the operating system can generate a disk with a set number of blocks, and also we need to FUSE framework so that we can use it to implement our filesystem in userspace. Since it is important to use both disk structure and superblock in the operating system, our team decided to introduce what each of these does in the operating system and we decided to write how we approached each section and how we implemented our codes into it.

Assumptions
We are assuming an integer is 32 bits (as it is on our systems). This means that when we write to the superblock and other areas where size is important, an integer will fit perfectly in the 4 byte word size.

Since the root directory does not have a directory entry for itself, we cannot specify the size of the directory. Thus, we allocate full blocks to the root directory, meaning if there are 18 entries (16 entries per block as blocks are 1024 bytes and each entry is 64 bytes), we allocate a whole two blocks and don't specify that we only use the first 128 bytes of the second block. This is fine since blocks are zeroed out when allocated, and a zero in the start block indicates an entry is unused.

We assume we will have a relatively small amount of open files in our open files list. Because of this, we perform O(n) traversals through our list to allocate a new file_handle and when any operations are performed on the open files list.

Disk Structure:

The assignment specifies that we need to divide a disk into 4KB blocks and each of these divided disk blocks should have read or written in its entirety. What we can verify now is that the disk is broken down into N number of 4kb blocks. The first block that has been divided from the disk (Block 0) is reserved as the Super Block, and the next k number of blocks (Blocks 1 to k) are reserved for the FAT blocks. Moreover, we can conclude that the value k is determined using N divided by 1024 FAT blocks. Here, the letter N represents the total number of disk blocks and 1024 represents how many 4 byte block pointers each block can hold. The remaining blocks are the data blocks and we can use those to store files and directories within the operating system. Since we know that the super blocks and FAT blocks are reserved, we can mark them as unused. Below shows a detail of this in a table format:

Block	Content
0	Super block
1 - *k*	File Allocation Table
k + 1 and higher	Data Blocks

Block Structure:

Super Blocks

It is important to mention what super blocks are. The Super Blocks consist of five 4 byte words that stores the metadata of a disk. According to the assignment 4 PDF that was provided to us, it says that the first word stores a magic number, 0xfadedbee. Also, this magic number is a unique identifier that can be used to identify what filesystem type that has been used. The second word, *N*, is for counting the total number of disk blocks on the disk. The third number, *k*, is used to identify the total number of FAT blocks or the value k. The fourth word indicates the size of each block. We are using 4KB blocks, so the size of each block will be set to 4096. Lastly but not least, the fifth word is used to show the starting block of the root directory.

Word Index	Content
0	Magic number: 0xfadedbee
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

<u>FAT Blocks</u>
File Allocation Table blocks follow next to the consecutive blocks in the super block. Each of these File Allocation Table block can store up to 1024 of 4 byte block pointers. In this block, the index of FAT each entry represents a block in the disk. The 4 byte value stored in each entry represents either the block's current status or its next block entry in a disk. According to the PDF, the value of 0 (zero) as an entry means a free block and -2 means that the block is the last one in the file. A 1 as an entry means that FAT entries should never be allocated for those blocks. Any other number other than those numbers mentioned above would represent the block's next block entry.

<u>Data Blocks</u>
Data blocks consist of both information about files and directory and also data for each file stored in a disk. According to the PDF, it says that "directory entries in this file system are 64 bytes long, and contain the following information:

The first 6 words store a null terminated string as the file or directory's name of up to 24 byte long. 1 word has 32 bits or 4 bytes. The next two words (8 bytes since 1 word has 4 bytes) store the file or directory's creation time. The next following two words store the modification time and the next two words coming up the length of the file or directory. The next word stores the start block number. The value of this start block number represents the block that stores its first file or directory entry. The next number coming after the start block number is a flag. The flag is used to indicate whether the entry made is a file or a directory. A 0 means it's a regular file; a 1 means it's a directory. The last word, word 15, indicates unused, or may be used for other purposes. We made a table to show this visually.

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

Assumption:
we can assume our integer is 32 bits. This means when we write to the superblock and other important areas that size matters, an integer will perfectly fit into 4 byte work size. Each FAT block can hold 1KB block pointer, so we need N/1KB FAT blocks.

How does the FAT work?
Say you have a file. This file has a directory entry, which contains a pointer to a start block. If you're doing a read, you can access the first 1024 bytes from the start block, but what if you need to read more than 1024 bytes? That is where the FAT comes in. You take the start block pointer and index into the FAT. The entry in the FAT is the pointer to the next block, thus your next 1024 bytes of your file. If you need more data, you repeat the process, using the current pointer to index into the FAT to get the next block pointer. If the entry in the FAT is -2, this means that your current block is the last block of the file.

Tracking Open Files
We have a file struct which contains the path of the file, the block number, a block buffer (for quick access to memory) of the block which contains its directory entry, and a file handle number. All open files are stored in a linked list of this struct type.

To keep track of which file handle numbers are available, we simply do a traversal through the list and find the lowest available number that is not in use (and is >2).

We use the file handle functionality of FUSE. This means when we open a file we set the file handle number, and refer to this when reading, writing, and closing a file. The file handles are used to retrieve an open file struct from the linked list.

If the directory entry is updated, we write the new entry back into the block buffer and then write the block buffer back to disk.

Caching Directory Block, Superblock, and FAT

To make our code to run speedy, we store our superblock and FAT in a memory buffer inside of our program. Additionally, we store the block which contains a given open files directory entry for quick modification of its fields. We use write-through writes to maintain consistency with our block device.

FUSE Install and Compiling Tutorial

Before we begin this assignment, we have downloaded FUSE from github to build and install FUSE libraries. We followed what was uploaded on Canvas and followed a step-by-step procedures of using FUSE. First download the version 2.9.6 release from the github repo the professor linked in the assignment details. Libfuse releases and places it on the FreeBSD kernel or we can use the following command on the FreeBSD machine:

```
curl -L https://github.com/libfuse/libfuse/releases/download/fuse-2.9.6/fuse-2.9.6.tar.gz -o fuse-2.9.6.tar.gz
```

After that, we need to extract it using tar command:

```
tar xvf fuse-2.9.6.tar.gz
```

Afterwards, type *cd* into the directory so like so:

```
cd fuse-2.9.6
```

Now run the following commands in order

```
./configure
make -j8
```

Use root mode then type: make install

If you have reached up to this moment, then you can compile and install the libraries that you needed to compile a FUSE programs. This will also compile all the examples that came with the FUSE.

```
sudo kldload fuse.ko
```

However, if you decided not to run this command after every reboot, then you can simply run the following and this will load the FUSE module on boot.

```
sudo echo 'fuse_load="YES"' >> /boot/loader.conf
```

Now you can restart the FreeBSD and you do not need to run kldload whenever you need to reboot.

RUNNING AN EXAMPLE

Running examples of FUSE is important because this will help you to understand how to run the examples using FUSE. First you need to go to the example directory inside the fuse-2.9.6 and you will see examples.

Type the following:

```
cd example/
```

And then create a directory that the filesystem will use as a mount point

```
clang -g -I/usr/local/include -D_FILE_OFFSET_BITS=64 -o hello.o -c hello.c

clang hello.o -L/usr/local/lib -lfuse -o hello
```

After this moment, we need to enter the root mode and create a new directory:

```
Mkdir mymount/
```

Note: the name of the new directory must be mymount otherwise the script test we made won't run

And now we have a new directory

```
./hello mymount/
```

When we enter that directory we could see a hello file, if we type "less hello", the information inside the hello will be printed out.

And there will be a hello executable in mountpoint directory.

```
ls mymount
```

You can also unmount a FUSE filesystem with the umount command like so

```
sudo umount mymount
```

Remember: everything should be executed under the root mode, otherwise you will get permission denied.

DEFINING FUSE OPERATIONS

When deciding which FUSE operation we had to implement, we can start work on the code. Our fuse program starts with parsing the device name from the argument list, and creating a new arguments list without the device name to be passed into fuse_main. We then initialize our block device handler, passing it our block device name. Next, we pass the modified arguments list into fuse_main, along with our struct of operators we implemented.Below is a name and a short description of each of the FUSE operations we implemented for this project. The implementations of each of these functions can be found in our pseudocode section of the design.

* **getattr(path, stat_buf):** This will call get_entry on the given filepath, then populate the stat struct with the blocksize of our filesystem, its total size in bytes, time of last access, modification, and creation (status) time. All unused fields are set to 0.
* **fgetattr(path, stat_buf):** We just call getattr.
* **access(path, mask):** This returns -ENOENT if the path is invalid, and 0 if the path is accessible. Since we don't handle permissions this doesn't have to be complicated.

* **readdir(path, buf*, filler, offset, file_info):** We call get_entries on the given path. This gives us a list of all directory entries in the directory. We take the first entry and create a struct stat for it. We then use filler to fill the buffer. Then we find the next file in the directory and repeat the steps of creating a stat struct, and using filler on it until there are no more directory entries in the directory to process. We also add the "." entry.
* **mkdir(path, mode):** Calls make_dir (from our handler) with the path.
* **unlink(path):** Calls unlink_file (from our handler) with the path.
* **rmdir(path):** Calls remove_dir (from our handler) with the path.
* **rename(from, to):** Calls rename_file (from our handler), which moves the directory entry over to the proper location.
* **utimens(path, ts[]):** We update the access and last modification times for the given file. We use the read_entry_at call to get the given directory entry. Then we modify it, and call write_entry_at.
* **open(path, file_info):** We call open_file (from our handler) on the file_path and are given a file handle. Then we set the file handle field in the file_info struct, and return the file handle.
* opendir(path, file_info): We chose not to implement this as directories are not cached like files are.
* **read(path, buf*, size, offset, file_info):** We extract the file handle from our file_info, and then call read_file from our handler, passing in the file handle, buffer, offset, and number of bytes to read. We return the number of bytes read.
* **write(path, buf*, size, offset, file_info):** Extracts file handle from file_info and passes the handle, buffer, size, and offset into write_file.
* **statfs(path, stat_buf):** We chose not to implement this as we do not have a way to store the number of free blocks and number of free files in the filesystem.
* **release(path, file_info):** Call close_file on the file handle extracted from the file_info.
* **releasedir(path, file_info):** We purposefully don't implement this because we don't track the open directories.
* **fsync(path, isdatasync, file_info):** All directory entry modifications are write-through, so no need to implement this.
* **fsyncdir(path, isdatasync, file_info):** We don't implement this as directories aren't cached.
* **flush(path, file_info):** We don't implement this as all writes are write-through and we don't have a cache for open files.

Automation (scripts)

We developed the following scripts to accomplish different tasks:
* Makefile: This allows us to build our FUSE executable.
* fs_mount: Given a block device name and a mountpoint, this creates a directory for the mountpoint, and then runs fat on the mountpoint to get FUSE running.
* fs_umount: This un-mounts the fileystem and then removes the directory.

Testing Filesystem

We developed unit tests for both our device handler and our FUSE program. The test files are called dev_test.c for the device handler test, and the FUSE test is called fat_test.c. To run dev_test execute "make dev_test", then run "./dev_test <disk name>". To run our FUSE tester, set the TEST flag at the top of fat.c to 1, then run "make", then execute "./fat <disk name>".

We created a script to automate the running of our device handler unit tests. Simply execute "./run_dev_test.sh".

In addition to the unit tests, we wrote a script to test regular filesystems commands. To run this execute "./run_command_test.sh".

Pseudocode & Implementation:
For this assignment, we have to create a C program that works similar to how we use FUSE examples and we had to create our version of file system. We are going to demonstrate how we wrote the code and how we implemented our knowledge into the C program.

We initiated building a C program that implements the system calls. We wrote a C program called fat.c and it contains what a simple shell program would work. In this version, we added more system calls and we made another version of shell program that can create a directory and open up a directory.

Pseudocodes here:

```
// initialize superblock
struct superblock{
 uint32_t magnum;
 uint32_t blocksFS;
 uint32_t blocksFAT;
 uint32_t startblock;
 uint32_t root;
};

//direct entry
struct direent{
 char filename[24];
 uint64_t ctime;
 uint64_t mtime;
 uint64_t atime;
 uint32_t unused;
};

//global variables and macros
static int dev_fd;
static int i;
static struct superblock super_block;
static int * FAT;
static struct file_list open_files;
static int diskImage;
uint32_t num_blocks;
uint32_t num_FAT_blocks;
time_t rawtime;
struct tm * timeinfo;

// FAT low API
int write_FAT (void);
int cache_FAT (void);
int write_superblock (void);
int cache_superblock (void);

// Middle class APIs
int in_block_range(uint32_t);
int get_free_block_num (void);
int allocate_block (void);
int clear_block (uint32_t);
int next_block (uint32_t);
int free_block (uint32_t);
int read_block (uint32_t, void *);
int write_block (uint32_t, void *);
int count_blocks_from (uint32_t);
int free_blocks_from (uint32_t);
int read_blocks_from (uint32_t, void **);
int write_blocks_from (uint32_t, void **);

// High core APIs
int read_entries_of (struct direent *, struct direent **);
int write_entries_of (struct direent*, struct direent **);
int forge_root_entry (struct direent *);
void read_entry (int, struct direent *);
void write_entry (int, struct direent *);

int get_N_block(struct direent *, int);
int get_entry_block(const char *);
```

We initialize retval and if that retval is unlinked with the path, then the system call redirects to a negative one and returns to errno. Later the function returns to zero.

```
static int fat_unlink(const char *path){
   int retval;

   retval = unlink(path);
   if(retval == -1) return -errno;

   return 0;
}static int fat_rmdir(const char *path){
   int retval;

   retval = rmdir(path);
   if(retval == -1) return -errno;

   return 0;
}
```

We need creates function pointers for our operations

Defining FUSE operations

```
fat_getattr (path, stat_buf):
  de = directory entry for path
  sb = the superblock
  fill_stat_buf(de, sb, stat_buf)
  return 0
```

```
// create disk image contains magic number, block size and number of FAT blocks
CreateDiskImage: (imageName, uint32_t num_blocks) {
  the magic_num = 0xfadedbee,  block_size = 1024, num_FAT_blocks =
(num_blocks/(block_size/4))+1, rootBlock = num_FAT_blocks + 1
  fd opens the imageName and write only.
                    If fd is not 1 then creat imagename, write only.

  write(fd, size of (&magic_num), 4);
  write(fd, size of (&block_size), 4);
  write(fd, size of array (&num_FAT_blocks), 4);
  write(fd, size of array (&num_blocks), 4);
  write(fd, size of array (&rootBlock), 4);

  //fill the rest of our superblocks
  void* emptyBuffer = malloc(block_size-(4*5));
  write(fd, emptyBuffer, block_size-20);

  //fill in the file allocation table
  int allocationTable[num_blocks];
```

```
 for (i=0; i < rootBlock; i++) {
   allocationTable[i] = -1;
   allocationTable[rootBlock] = -2;
 }
 for i is zero and until it reaches rootBlock, i++) {
   allocationTable[i] = -1; (error)
 }
 allocationTable[rootBlock] = -2;
 for i is 0 and i is less than rootBlock + 1; num_blocks++) {
   allocationTable[i] = 0;
 }
 write(fd, allocationTable, num_blocks);
 // Fill in rest of last block in file allocation table
 uint32_t *buffer = malloc(sizeof((num_FAT_blocks * block_size)-num_blocks));
 for(i=0; i < sizeof(buffer); i++){
   buffer[i] = -1;
 }
 write(fd, buffer, sizeof(buffer));
 for (int i = 0; i < rootBlock; num_blocks++)
 {
   write(fd, emptyBuffer, block_size);
 }
}
```

In this function, we can remove a directory by using rmdir system call that is built into the C programming language. It works similar like *fat_unlink* function.

```
fat_fgetattr(path, stat_buf):
  return myfs_getattr(path, stat_buf)

fat_access(path, mask):
  if dir_entry exists for path: return 0
  else: return -ENOENT

fat_readdir(path, buf*, filler, offset, fi):
  DIR *dr;
  struct dirent *d;

  (void) offset;
  (void) fi;
  dr = opendir(path);
  // struct stat st;
  // memset(&st, 0, sizeof(st));
  if (dr == NULL)
          return -ENOENT;
  filler(buf, ".", NULL, 0);
  filler(buf, "..", NULL, 0);
  int i = 0;

  while((d = readdir(dr)) != NULL){
    struct stat st;
    memset(&st, 0, sizeof(st));
    st.st_ino = d->d_ino;
    st.st_mode = d->d_type << 12;
    filler(buf, d->d_name, &st, 0);
  }

  closedir(dr);
  return 0;
}

fat_mkdir(path, mode):
int retval;

  retval = mkdir(path, mode);
  if(retval == -1) return -errno;
  int i = 0;
  return 0;


fat_rmdir(path):
             int retval;
                retval = rmdir(path);
                   if(retval == -1) return -errno;
                          return 0;
}
```

```
fat_rename(from, to):
        Initialize integer retval
        Retval uses rename system call and then return to error if the file does not exit.

fat_truncate(path, size):
        Initialize integer retval
        Retval uses a system call called truncate to truncate on free BSD
```

```
fat_ftruncate(path, size):
        (works same as above)

fat_utimens(path, ts[]):
  de = dir entry of path
  de->acc_time = ts[0]
  de->mod_time = ts[1]
  set_entry(path, de)

fat_open(path, fi):
  file_handle = open_file(path)
  if file_handle == -1: return -1
  fi->fh = file_handle
  return file_handle

fat_read(path, buf*, size, offset, fi):
  return read_file(fi->fh, buf, size, offset)
```

Starts with a fi. Assign retval with pread(system call) (fi points to fh, buf, size and offset)
If the value of retval is negative one, then print out error and then return to retval.

```
fat_write(path, buf*, size, offset, fi):
  return write_file(fi->fh, buf, size, offset)

fat_release(path, fi):
  return close_file(fi->fh)

static struct fuse_operations fat_oper = {
  .getattr    = fat_getattr,
  .access     = fat_access,
  .readdir    = fat_readdir,
  .mkdir      = fat_mkdir,
  .unlink     = fat_unlink,
  .rmdir      = fat_rmdir,
  .rename     = fat_rename,
  .truncate   = fat_truncate,
  .ftruncate  = fat_ftruncate,
  .utimens    = fat_utimens,
  .open       = fat_open,
  .read       = fat_read,
  .write      = fat_write,
  .release    = fat_release,
  .create     = fat_create,
};
```

```
main(args):
  dev_name = args[1]
  init_handler(dev_name)

  fuse_args = args without args[1]
  fuse_main(fuse_args, myfs_oper)
```

Testing
We should have several testing files to test whether our program works or not. First we have a script file called testing.sh,

```
//-----------------------------------------
cd mymount
ls
mkdir test_dir
ls
cd test_dir
ls -la
cd ..
rmdir test_dir
ls
//-----------------------------------------
```

For testing the main program, we need several different functions:

1. Testing whether we can get the superblock

2. Testing to read and write in a entry

3. Testing to make a new directory
    1) Confirm the path does not exist
    2) Read in the entry
    3) Create directory entry for the new path
    4) Test the new directory is valid right now

4. Testing whether the path is valid
    1) Verify we can get the path and filename

5. Testing whether we could delete a directory
    1) Create a new directory which does not exist
    2) Verifying the directory now exist
    3) Add file to directory

4) Exit the directory and try to delete the directory (which should not work)
5) Enter the directory and delete the file
6) Exit the directory and remove the directory
7) Verify the directory does not exist

6. Testing whether we could delete a file
    1) Create a empty file
    2) Verifying the file now exist
    3) Remove file
    4) Verifying the file now gone

7. Testing whether we could rename a file
    1) Create a new file
    2) Verify the file exist
    3) Store entry of the original file
    4) Then verify the new file name does not exist anymore
    5) Change file to new file name
    6) Verify the original file name does not exist
    7) Verify the new name exist
    8) Store entry of new file

8. Testing whether we could some complex argument.
    1) Open a file
    2) Verifying that the entry was written and matches
    3) Make sure the size is zero
    4) Write to a file
    5) Make sure the entry was written and matches again
    6) Verify the size of bytes
    7) Read back to a file
    8) Read buffer
    9) Close the file afterward

Acknowledgement:
Erik, Seongwoo, Michael, and Yuzhuang would like to thank everyone who was involved in Professor Darrell Long's CMPS 111 - 01. Thank you to the graders and every member of our team who worked so hard to finish every assignment.