DESIGN ASSGN4

Nikola Panayotov (npanayot@ucsc.edu)
Marko Jerkovic (mjerkovi@ucsc.edu)
Edwin Ramirez (edalrami@ucsc.edu)
Jeremiah Liou (jliou@ucsc.edu)

Fs_init.c():

FS_Init is a short file used to generate the disk image for our file system. It takes in exactly two arguments, the first being the name of the disk, and the second being the size in bytes for the disk. For example, to create a disk called “myDisk” with a size of 12 MB, the command ‘./fs_init myDisk 12582912’ would be used.

File_system.c():

The implementation of the file system requires writing the FUSE API functions as well as, creating helper functions, and actually writing a main().

Structs:

Superblock:

Superblock holds 5 32-bit unsigned integers to contain the information, of the superblock: the magic number, number of blocks(n), number of FAT blocks (k), the size of a block (4096), and the starting location of the root block (k+1).

Directory:

The directory struct contains the information stored in a directory entry. A 24 bit char array is used to hold the file name, 3 64-bit unsigned ints are used to hold the creation, modification, and access times of the file or directory, and 4 32-bit unsigned ints are used to hold the length in bytes, the start block, the flag to determine if the entry is a directory of regular file, and the final 4 bytes to be left unused.

Helper functions:

`int get_disksize(const char* filename):`

This function opens the disk file with read/write permissions, and obtains the stats from it using statfs(). The st_size parameter is returned, giving the total size of the disk.

`void superblock_init(int fsize, int fd):`

Superblock_init takes in two arguments: the size of the disk and the disk’s file descriptor. A superblock struct is declared and populated with the magic number, number of blocks (size of disk divided by the block size), the number of blocks in the fat (number of blocks / 1024), the specified block size, and the start of root. This superblock is then written to the first sizeof(superblock) bytes in the disk.

`fat_init(int fd):`

Fat_init initializes the FAT. The superblock is first read in to get N and K, the number of blocks and number of FAT blocks, respectively. An array of 4 byte signed integers of length N is initialized. The entries 0 through k are set to -1 to represent the superblock and FAT. The k+1 block stores the root directory and starts of with only a single entry so a -2 is set in its index. Finally, the remaining entries from k+2 to N-1 are set to 0, indicating that those blocks on the disk are free to use.

`dir_init(int fd, int block, uint32_t dot, uint32_t dotdot):`

dir_init takes in a file descriptor, the block to initialize, a 32-bit unsigned int for the "." entry, and another 32-bit unsigned int for the ".." entry. The block and integer for the "." entry should be the same. Two directory structs "." and ".." are populated with all relevant information. Next, the data block given from the block argument is seeked to, and the "." and ".." entries are written sequentially into the data block as the first and second entries there. Finally, the FAT entry corresponding to the newly written directory is set to -2, indicating that the associated data block is currently in use.

`root_init(int fd):`

Root_init seeks to the k+1 data block, and calls dir_init with the block, "." and ".." arguments all being the same, k+1 to initialize the data block for root.

`find_path(const char *path, int block_num, directory *dir_entry):`

Find path implements the logic to traverse through the FAT file system to determine if a path exists. It takes in a char * for the path to search, an integer to start the search from (usually the root block since FUSE always searches for an absolute path), and finally a pointer to a directory struct in order to pass by reference the directory entry of the data block to find. The return value of the function is the the block number where the desired path ends, or -1 if the path does not exist.

Given the path name for a file and the block number of where to begin the search, the path is parsed into individual path tokens, and recursively looks through the entire path by calling find_path() with the truncated path and the block number of the current directory. For instance, if the path to search was '/foo/bar/quux' and the file foo was found in the root with start block 3001, the recursive call would be find_path("/bar/quux", 3001, temp_dir). If the file quux is found, the start block of quux is returned, and the metadata is returned by reference in the directory pointer argument.

```
add_dir_entry(int fd, char *fname, int32_t parent_blk, int32_t child_blk,
int is_dir):
```

This function adds an entry to the block of directory entries of its parent directory. It takes in an int file descriptor, the name of the directory entry to add, the block number of its parent block, the block number of the child block to add (AKA itself), and a flag to specify whether the newly added entry is a directory or a regular file.

First, the last FAT entry of the parent directory is obtained from searching the FAT, that is, the FAT is searched, starting at the parent's start block, until the index containing -2 is found for that FAT block chain. A directory struct is initialized with the appropriate metadata, including the filename argument, and the is_dir argument. With the directory entry set up, the parent directory block is scanned to find the first empty directory slot, and the FAT entry of the child block is updated to signify that its corresponding data block is in use. If the parent directory block had no free spaces (all 64 entries were filled), then the directory is expanded by finding the next available data block, setting the corresponding FAT entry, and adding the new child entry to the start of the new parent data block.

```
find_dir_entry(int fd, int32_t start_blk, char *entry_name,
int32_t *actual_block):
```

This function searches the entire FAT block chain of a given data block of directories for a specific entry name. The FAT index of the data block in which the entry resides in is returned, and the index of the desired entry is returned by reference.

The start block of the directory to search is seeked to, and all of its entries are searched, to locate the desired entry. If none of the 64 entries match the name of the entry to find, then the contents of the corresponding FAT block are parsed. If the FAT block was -2, then it means the desired entry does not exist, and error is returned. If the FAT block is a positive number, then that index is saved, and the associated data block is searched. This loop continues until either a match is obtained, or the entire block chain is searched with no match.

FUSE Functions:

```
fat_getattr(const char *path, struct stat *sbuf):
```

The getattr function uses find_path on the path argument first. If find path returns an error, then the entry does not exist, and the appropriate error is returned. If fidd path returns the block number of the desired path, then the temporary directory passed by reference from find path is saved into the sbuf struct to be passed by reference, and a success value is returned.

`fat_mkdir(const char *path, mode_t mode):`

mkdir first checks to see if the directory to be made already exists in the path using find_path. If find_path returns a valid value, then fat_mkdir returns an error. If the directory does not exist, the path is parsed into a prefix (the path in which the directory should be placed), and a suffix (the new directory name). Find_path is used on the prefix to return the starting block of the parent directory, and add_dir_entry is called to add the new directory to the parent's block of directories. The FAT needs to also be updated, so the first 0 value in the FAT is searched for, set to -2, and dir_init is called to initialize its associated data block.

`fat_unlink(const char *path):`

Unlink is used to delete a file. The path is parsed into a prefix and a suffix like in mkdir, and find_path is used to determine if the file exists or not. If the file does not exist, or is a directory, unlink cannot be used to remove it so an error is returned. If the file exists and is a regular file, the data block number for its parent directory is obtained, its FAT chain is traversed, jumping to each data block based on the index of the FAT entry, and all entries in the data block are iterated over, looking for a match. If a match is ever found, its start block is saved, and used to clear both the file's data block, as well as its FAT entry. If the entire FAT chain is traversed with no match, then error is returned.

`fat_create(const char *path, mode_t mode, struct fuse_file_info *fi):`

Create is used to create a regular file. The path is parsed into a prefix and a suffix, and the prefix is checked to see if the file already exists there. If the file already exists, then return an error since it is not possible to make a file with a name already in use. Otherwise, the FAT is searched for a free FAT block and is set to signify that the data block is currently in use, and add_dir_entry is used to update the parent directory with the new file entry.

```
fat_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t
offset, struct fuse_file_info *fi):
```

Readdir is used to read an entire directory. Find_path is used to return the location of the directory to read first. If the directory entry's flags are set to indicate that it is a file, then an error is returned. Otherwise, the entire FAT block chain of the file is traversed, and each data block is scanned to save every valid entry inside of it into buffer. Once the entire FAT chain is iterated through, and all entry blocks are scanned, a success is returned.

```
fat_write(const char *path, const char *buf, size_t size, off_t offset,
struct fuse_file_info *fi):
```

Write parses the path argument, and checks if the file exists first and that it is not a directory. Next, the function checks if more blocks need to allocated in order to be able to write size bytes starting at the offset. If more blocks are needed, then the number of blocks to be added is calculated, and the FAT is traversed, attempting to allocate that many blocks. If the there are not enough blocks to allocate, then an error is returned. Otherwise, the newly-allocated blocks are added to the FAT block chain of the file, and the indices of all of the FAT blocks for the file are cached, so that write can be done quickly. The actual location of where to start the write operation is given to by offset. It should be noted that for every 4096 bytes in the offset, the starting data block to write to is incremented. WIth the starting location determined, we then write as many bytes as possible to the current starting data block, either until the block is full (reached byte 4096), or the number of bytes written = size. If the number of bytes written = size, then we return the number of bytes written, indicating a success. Otherwise, the next data block (given by the cache of FAT indices) is seeked to, and is written to. This repeats until bytes written = size.

```
int fat_truncate(const char *path, off_t size):
```

```
int fat_utimens(const char *path, const struct timespec ts[2]):
```

```
int fat_chown(const char *path, uid_t uid, gid_t gid):
```

```
int fat_open(const char* path, struct fuse_file_info* fi):
```

Open simply checks if a file exists, and returns a success if it does, and an error if it doesn't. The fuse_file_info struct is also populated, to be passed back by reference.

```
int fat_read(const char *path, char *buf, size_t size, off_t offset, struct
fuse_file_info *fi):
```

Like most other previous functions, read first checks if the file exists and that it is not a directory. It then traverses the path's FAT block chain, caching the FAT indices in order to more quickly read, similar to the write operation. The location/data block from where to start reading is determined by the offset, and once the reading location is seeked to, 1 byte is read in at a time, until either the entire data block is read, EOF is encountered, or the number of bytes read = size. If the entire data block is read, then the next data block given by the cached array of FAT indices is seeked to and the read continues. This reading process loops until number of bytes read = size, or EOF is read. It returns the number of bytes read on success.

```
fat_rename(const char* from, const char* to):
```

In the rename, we first parse both from and to, to get what we want to do with them. We then check if the prefix exists, and if it does not exist then we can exit out with an error. Otherwise, we check if the "to" exists to see what to do. If it is not found in the "to" place, then we can just move the file there. If it is found in the to place, then we replace it. FUSE already handles whether or not it is a directory or file so we can just focus on putting the data at the correct place.

```
static int fat_rmdir(const char *path):
```

The rmdir function makes the assumption that the directory to remove is empty, meaning that its only contents are "." and "..". As before, find_path is used to check that path actually exists first. Next, we navigate to the data block of the directory to be removed, and check its length. If the length is more than the size of 2 directory entries (128 bytes), then this means that the directory is not empty, and therefore cannot be removed. The parent of the directory to remove is then scanned to find the entry for the directory to be removed, and its start block is set to 0. The corresponding FAT entry of the directory to be removed is cleared to 0. Finally, the data block of the directory is seeked to again, and since it is known that it only has 2 entries, the "." and ".." entry start blocks are set to 0.

Testing:
To test the file system we created directories and files in dummy which is our root directory. We then also added more sub-directories to the directories in the root. To test that file creation and writing was working, we redirected an echo to a file that doesn't exist:

```
Echo "Some string" > foo
```

To check if cat was working we would cat the newly created file. To check if mv worked, we would create a directory /foo with a file bar in it. We would then create a file bar in root and do:

```
Mv bar /foo
```

This would replace the bar in foo with the bar from root and delete the bar in foo.

We also tested unlink by calling rm on normal files. We used the rmdir terminal command to test fat_rmdir on empty directories.