Lab 3 - file system

Programming Assignment 3 - simple file system

In this assignment you will implement a simple Unix-like file system using the FUSE library. This is the first posting; additional ones will give implementation hints and describe test strategies.

Preparation

You will need to install the development headers and libraries for FUSE:

```
sudo apt install libfuse3-dev
```

Assignment Details

Materials: You will be provided with the following materials in your team repository:

- Makefile
- `fs5600.h` - structure definitions
- `homework.c` - skeleton code
- `misc.c`, `hw3fuse.c` - support code
- `mkfs.py` - utility to format disk image

Deliverables: You will be using the FUSE library, which is based on the VFS interface, and will need to implement the following methods:

- `getattr` - get attributes of a file/directory
- `readdir` - enumerate entries in a directory
- `create` - create a file
- `read`, `write` - read data from / write to a file
- `unlink` - delete a file
- `truncate` - delete file contents but not the file itself
- `mkdir`, `rmdir` - create, delete directory
- `rename` - rename a file (only within the same directory)
- `chmod` - change file permissions
- `init` - constructor (i.e. put your init code here)

You'll pass FUSE a structure with pointers to these functions, and it will call them as necessary to handle file system operations. Note that return values follow a fairly common Unix standard - negative values are error numbers, and non-negative values are successful return values.

You will implement a nearly fully-featured Unix file system, with inodes, allocation bitmaps, and indirect blocks. You **won't** have to implement:

- "holes", also known as sparse files
- renaming across directories
- files larger than 64MB
- truncating to a non-zero length
- long filenames - file names are limited to 27 bytes plus a terminating 0
- permissions - you need to store permission information so the OS can use it, but you don't need to check it yourself.
- ownership - FUSE is weird about that. We'll just set `uid` and `gid` to zero, because Linux will assume your user ID owns all the files in the file system anyway.

Note that your code will **not** use standard file system functions like `open`, `read`, `stat`, `readdir` etc. - your code is responsible for files and directories which are encoded in 1024-byte data blocks which you access only via the `block_read` and `block_write` functions from `misc.c`.

You're given the following helper functions:

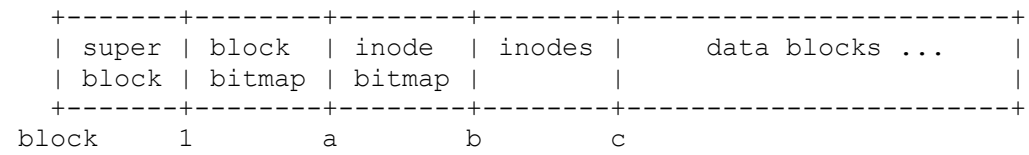
- `bit_test`, `bit_clear`, `bit_set` - manage in-memory bitmap
- `parse_path` - modeled after `parse` in homework 1

File System Format

The file system uses a 1KB block size; it is a simplified version of `ext2` with allocation bitmaps and inodes grouped at the beginning of the disk instead of being spread out in block group. Inodes have 6 direct pointers, an indirect pointer, and a double-indirect pointer, for a maximum of 64K+256+6 blocks in a file.

The disk is divided into 5 regions:

1. the superblock, which tells you how big the regions are
2. the block bitmap, for allocating blocks
3. the inode bitmap
4. the inode region, containing 64-byte inodes packed 16 to a 1024-byte block
5. data blocks for directories, file data, and indirect blocks



An empty image (created with the provided `mkfs` utility) has:

- an empty root directory, in inode 1
- blocks used by the superblock, bitmaps, and inodes are marked "in use" in the block bitmap
- inode 0 is marked "in use" in the inode bitmap

Superblock: The superblock is the first block in the file system, and contains the information needed to find the rest of the file system structures. The following C structure (found in `fs5600.h`) defines the superblock:

```
    struct fs_super {
              int32_t magic;             /* 0x37363030 ('5600') */
              int32_t disk_size;         /* in 1024-byte blocks */
              int32_t blk_map_len;       /* block map, in blocks */
              int32_t in_map_len;        /* inode map, in blocks */
              int32_t inodes_len;        /* inode table len, in blocks */
              char pad[1004];            /* to make size = 1024 */
    };
```

Note that `int32_t` is a standard C type found in the `<stdint.h>` header file, and refers to an signed 32-bit integer. (similarly, `uint16_t`, `int16_t` and `uint32_t` are unsigned/signed 16-bit ints and unsigned 32-bit ints)

Hint - have a global variable of type `struct fs_super`, and read the superblock into it at startup so you'll have access to these. Maybe also calculate the starting block of the inode map and inode region. (the block map always starts at block 1, of course)

Another hint - you might want to keep copies of the two bitmaps and the inode region in memory. That will make it easy to access them when you implement the first (read-only) part, and later you can modify them in memory and then write them back to the disk.

Inodes: These are similar to the `ext2` inodes discussed in class and the HOSW book. Each inode corresponds to a file or directory; in a sense the inode **is** that file or directory.

```
struct fs_inode {
    int16_t uid;      /* file owner */
    int16_t gid;      /* group */
    int32_t mode;     /* type + permissions (see below) */
    int32_t mtime;    /* modification time */
    int32_t size;     /* size in bytes */
    int32_t ptrs[6];
        int32_t indir_1;
        int32_t indir_2;
        int32_t pad[4];   /* to make it 64 bytes */
};
```

"Mode": The FUSE API (and Linux internals in general) mash together the concept of object type (file/directory/device/symlink…) and permissions. The result is called the file "mode", and looks like this:

```
        |<-- S_IFMT --->|           |<-- user ->|<- group ->|<- world ->|
        +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        | F | D |   |   |   |   |   | R | W | X | R | W | X | R | W | X |
        +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Since it has multiple 3-bit fields, it is commonly displayed in base 8 (octal) - e.g. permissions allowing RWX for everyone (rwxrwxrwx) are encoded as '777'. Note that in C an octal number is indicated by putting a spurious 0 in front (e.g. 0777); in Python it's 0o777.

Directories: Directories are a multiple of one block in length, holding an array of directory entries:

```
struct fs_dirent {
    uint32_t valid : 1;
    uint32_t inode : 31;
    char name[28];       /* with trailing NUL */
};
```

Each "dirent" is 32 bytes, giving 1024/32 = 32 directory entries in each block. The directory size in the inode is always a multiple of 1024, and unused directory entries are indicated by setting the 'valid' flag to zero. The maximum name length is 27 bytes, allowing entries to always have a terminating 0 byte so you can use `strcmp` etc. without any complications.

what's that ": 1" and ": 31" thing? It combines two structure fields into a single integer - 1 bit for the valid flag, and 31 bits for the inode number.

Storage allocation:

Inodes and blocks are allocated by searching the respective bitmaps for entries which are cleared. Note that when the file system is first created (by mktest or the mkfs-hw3 program) the blocks used for the superblock, maps, and inodes are marked as in-use, so you don't have to worry about avoiding them during allocation. Inodes 0 and 1 are marked, as well.

The following functions are provided at the top of homework.c to access these bitmaps:

```
bit_set(map, i);
bit_clear(map, i);
bit_test(map, i);
```

where `map` is a pointer to the memory containing the bitmap and `i` is the index to set, clear, or check.

Path translation:

You're going to be doing a lot of it. You're given a function which splits a path into an `argv`-like array, much like

Functions to implement

For more information on FUSE, see CS135 FUSE Documentation from Geoff Kuenning at Harvey Mudd, although it's a few FUSE versions out of date now.

The FUSE interface is shown below, skipping a few of the methods we don't use. We'll create a bunch of functions (naming them `lab3_init`, `lab3_getattr`, etc.) then define a variable of type `struct fuse_operations` and fill in its fields with pointers to these functions, and pass a pointer to that structure to FUSE. We're going to ignore some of the arguments, which I've replaced with `...` here.

```
struct fuse_operations {
    void *(*init)(<stuff>);

    int (*getattr)(const char *path, struct stat *sb, ...);

    int (*readdir)(const char *path, void *ptr, fuse_fill_dir_t filler,
                off_t offset, ...);

    int (*read)(const char *path, char *buf, size_t len, off_t offset, ...);

    int (*mkdir)(const char *path, mode_t mode);
    int (*rmdir)(const char *path);

    int (*create)(const char *path, mode_t mode, ...);
    int (*unlink)(const char *path);
    int (*rename)(const char *src_path, const char *dst_path, ...);
    int (*chmod)(const char *path, mode_t new_mode, ...);

    int (*truncate)(const char *path, off_t new_len, ...);
    int (*write)(const char *path, const char *buf, size_t len, off_t offset,
...);
};
```

return values: the `init` function returns NULL. All other functions return:

- success: number of bytes read/written (`read`, `write`)
- success: 0 (all other methods)
- failure: negative error code

The error codes you'll return, along with their description (from `/usr/include/asm-generic/errno-base.h`):

- `ENOENT` - "No such file or directory"
- `EISDIR` - "Is a directory" - e.g. calling `read` on a directory
- `ENOTDIR` - "Not a directory" - e.g. `/dir/file.txt/xyz`
- `ENOTEMPTY` - "Directory not empty" - `rmdir` error
- `EEXIST` - "File exists" - error for `create/mkdir/rename`
- `ENOSPC` - "No space left on device"
- `EINVAL` - "Invalid argument" - for cases we don't handle

Note that you'll always return a negative error code, e.g. `return -ENOENT;`

Detailed method descriptions

Note that a bunch of these functions have a `struct fuse_file_info` argument, and some others have flag arguments. We'll ignore all of these.

```
void *lab3_init(struct fuse_conn_info *conn, struct fuse_config *cfg);
```

Gets called at startup. Ignore the arguments; you should probably read the superblock, and if you're going to keep copies of the bitmaps and inode table, you should read them too. (you'll need to allocate memory dynamically for them, but you can have global variables pointing to them)

Return NULL to make the compiler happy.

getattr:

```
int lab3_getattr(const char *path, struct stat *sb, struct fuse_file_info
*fi);
```

For a full description of `struct stat` see "man 2 stat"; to translate inode fields into struct stat:

```
    memset(sb, 0, sizeof(*sb));
    sb->st_mode = in->mode;
    sb->st_nlink = 1;
    sb->st_uid = in->uid;
    sb->st_gid = in->gid;
    sb->st_size = in->size;
    sb->st_blocks = div_round_up(in->size, BLOCK_SIZE);
    sb->st_atime = sb->st_mtime = sb->st_ctime = in->mtime;
```

readdir:

```
typedef int (*fuse_fill_dir_t) (void *ptr, const char *name,
                                const struct stat *stbuf, off_t off,
                                enum fuse_fill_dir_flags flags);
int lab3_readdir(const char *path, void *ptr, fuse_fill_dir_t filler, off_t
offset,
                 struct fuse_file_info *fi, enum fuse_readdir_flags flags);
```

C doesn't have an iterator type, so FUSE passes a function to `readdir`, and you call that function once for each name in the directory. In particular:

- Ignore the `offset`, `fi` and `flags` arguments.
- For each directory entry, call `filler(ptr, name, NULL, 0, 0);` where `ptr` is the argument passed to `readdir` and `name` is a pointer to the name field in the directory entry.

read:

```
int lab3_read(const char *path, char *buf, size_t len, off_t offset,
              struct fuse_file_info *fi);
```

Starting at `offset` bytes into the file, read `len` bytes (or less, if you hit end-of-file) into buffer `buf`. Ignore `fi`.

The preceding 4 methods are the read-only ones; when you're done with them you should be able to mount a prefabricated disk image, look around, and do some testing.

mkdir, rmdir:

```
int lab3_mkdir(const char *path, mode_t mode);
```

Create a directory named `path`. The containing directory must exist (return `-ENOENT` if it doesn't), and `path` must not already exist (return `-EEXIST` if it does). Because FUSE is strange sometimes, you need to set the `mode` field of the new inode to `mode | S_IFDIR`.

```
int lab3_rmdir(const char *path);
```

What it says. But check to see if the directory is empty first, and return `-ENOTEMPTY` if it isn't.

create, unlink:

```
int lab3_create(const char *path, mode_t mode, struct fuse_file_info *fi);
```

Create a zero-length file with name `path` and mode equal to `mode | S_IFREG`. ("REG" = "regular file") If the name is already in use, return `-EEXISTS`; if the directory to create it in doesn't exist, return `-ENOENT`.

```
int lab3_unlink(const char *path);
```

Delete a file.

write:

```
int lab3_write(const char *path, const char *buf, size_t len, off_t offset,
               struct fuse_file_info *fi);
```

Write `len` bytes from `buf` to the file, starting at `offset`. If `offset` is greater than the length of the file, return `-EINVAL`.

rename:

```
int lab3_rename(const char *src_path, const char *dst_path, unsigned int
flags);
```

Rename `src_path` to `dst_path`. If they're not in the same directory, it's ok to return `-EINVAL`. If `dst_path` exists and is a file, delete it first. If it exists and is an empty directory, delete it first; if it's a non-empty return `-ENOTEMPTY`.

chmod:

```
int lab3_chmod(const char *path, mode_t new_mode, struct fuse_file_info *fi);
```

Set `mode` to `(old_mode | S_IFMT) | mode`. (i.e. only set the bottom 9 or so bits)

truncate:

```
int lab3_truncate(const char *path, off_t new_len, struct fuse_file_info
*fi);
```

Truncate file to `new_len` bytes - feel free to return `-EINVAL` if `new_len > 0`. Ignore `fi`. (note that you'll need to free the blocks, and you might want to factor the logic in this function, as you'll need to do the same thing in `unlink`)

Error codes (and success return values) in Lab 3

An important part of this assignment is correctly implementing error codes when your methods are given invalid inputs.

One thing to remember - error codes are for when **someone else** made an error, e.g. your method is being called with a bad value. When you detect **your own** error, you should assert rather than returning an error, so that you can debug it.

Path translation: `ENOTDIR`, `ENOENT`

Path translation is a series of steps of lookup up a name in a directory. At each step you need to:

1. validate that the directory is, in fact, a directory
2. find the corresponding path component in that directory

For step 1 you need to check the `mode` field in the corresponding inode, and return `-ENOTDIR` if it is not:

```
if (!S_ISDIR(inodes[inum].mode))
    return -ENOTDIR;
```

For step 2, you need to iterate through all the blocks in the directory, searching each of them for the path component. If you don't find it, return `-ENOENT`.

Note that you can assume that there are no more than `N_DIRECT` blocks in a directory, so you only need a single `for` loop to iterate over them. (and a nested inner `for` loop to iterate over the directory entries in a block)

HINT: it will be easier if you factor this search into a function that looks something like `lookup(int dir_inode, char *name)`, returning an inode number or a (negativer) error number. When you find a match you can break out of the inner loop with `return`, and then return `-ENOENT` after the end of the outer loop.

Allocating blocks and inodes: `ENOSPC`

If you run out of blocks or inodes, your FUSE method should return `-ENOSPC`. This can be the case in `mkdir`, `create`, or `write`.

HINT: write allocation functions that return an integer block or inode number, and a negative error number if out of space.

Creating files and directories: `ENAMETOOLONG`

All path components in our file system have to be 27 bytes long or less, so that they can fit in a 28-byte directory entry field with a terminating null character.

Use `strlen` to check the length of the final path component in `create` and `mkdir`, and if it's over 27, return `-ENAMETOOLONG`. If you forget to do this, then an oversized file name will overwrite the next entry in the directory and break things.

HINT: when you're reading directories, you may want to `assert` that `strlen(de[i].name) < 27` or similar. Don't return an error code - see note at top.

Full list of error codes:

path translation errors = `ENOTDIR`, `ENOENT`

`getattr` - path translation errors. Returns 0 on success.

`readdir` - path translation, plus `ENOTDIR` if the final translation result ("leaf") isn't a directory. Returns 0 on success.

`read` - path translation, plus (optional) `EISDIR` if path is not a file. (FUSE will never invoke `read` or `write` on a directory) Returns number of bytes read on success. (=0 if end of file)

`rmdir` - path translation, plus `ENOTDIR` if path isn't a directory. Note: you can assume FUSE will never call `rmdir("/",...)`. Returns 0 on success.

`mkdir` - path translation if parent doesn't exist, `ENOTDIR` if parent is not a directory, `EEXIST` if the path exists, `ENAMETOOLONG` if leaf name is too long, `ENOSPC` if out of inodes or blocks. Returns 0 on success.

(I'm assuming you allocate a single block and set size=1024 when you create a directory)

`create` - path translation if parent doesn't exist, `ENOTDIR` if parent is not a directory, `EEXIST` if the path exists, `ENAMETOOLONG` if leaf name is too long, `ENOSPC` if out of inodes. Returns 0 on success.

`unlink` - path translation, plus `EISDIR` if path isn't a file. Returns 0 on success.

`truncate` - path translation, `EISDIR` if it's not a file, (optional) `EINVAL` if `new_len != 0`. Returns 0 on success.

`rename` - path translation, `ENOENT` if source path **doesn't** exist, `EEXIST` if destination path **does** exist, (optional) `EINVAL` if source directory is not the same as destination directory. Returns 0 on success.

`chmod` - path translation. Returns 0 on success.

`write` - path translation, `ENOSPC` if out of blocks or inodes, (optional) `EISDIR` if not a file. Returns number of bytes written on success. (which - on success - should always be number of bytes passed to your method)