Programming Project 3 - read-write file system

In this assignment you will implement a read-write version of a Unix-like file system using the FUSE (File system in User SpacE) library. Like its name FUSE allows us to implement a file system in the user space which makes the development much easier than in the kernel space. FUSE exposes a few file system function interfaces that need to be instantiated and your task is to fill in these function bodies. Until you have fully implement a certain set of functions (e.g., functions in the first part) you won't be able to actually mount and test your file system. Thus, you must write and run unit tests as you implement each function. As always, first read through the entire project description before you start working on the code.

1. Assignment Details

Materials: You will be provided with the following materials:

- Makefile
- fs5600.h - structure definitions
- homework.c - skeleton code
- unittest-#.c - test skeleton
- misc.c, hw3fuse.c - support code
- gen-disk.py, disk1.in - generates file system image

Deliverables: There are two parts to this assignment.

In the **first part,** you will need to implement the following functions in `homework.c`:

- `fs_init` - constructor (i.e. put your init code here)
- `fs_getattr` - get attributes of a file/directory
- `fs_readdir` - enumerate entries in a directory
- `fs_read` - read data from a file
- `fs_statfs` - report file system statistics
- `fs_rename` - rename a file
- `fs_chmod` - change file permissions

In the **second part,** you will need to implement these methods:

- `fs_create` - create a new (empty) file
- `fs_mkdir` - create new (empty) directory
- `fs_unlink` - remove a file
- `fs_rmdir` - remove a directory
- `fs_truncate` - delete the contents of a file
- `fs_write` - write to a file
- `fs_utime` - change access and modification times

(the actual function names don't matter - FUSE will call the function pointers in the `fs_ops` structure, which is initialized at the bottom of the file)

LIMITATIONS You can make the following assumptions:

1. Directories are not nested more than 10 deep. You don't need to enforce this - the test scripts will never create directories nested that deep.
2. Directories are never bigger than 1 block
3. `rename` is only used within the same directory - e.g. `rename("/dir/f1", "/dir/f2")`. The test scripts will never try to rename across directories
4. Truncate is only ever called with len=0, so that you delete all the data in the file.

Your code will run under two different frameworks - a C unit test framework (libcheck), and the FUSE library which will run your code as a real file system. In each case your code will read blocks from a "disk" (actually an image file) using the `block_read` function.

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 the data blocks which you access via `block_read` and `block_write`.

You will be graded on the following code and their execution:

- `homework.c` - implementation
- `unittest-1.c`, `unittest-2.c` - unit tests

Note that you will probably need to install several packages for the included code to work; in Ubuntu you can use the following commands:

```
sudo apt install check
sudo apt install libfuse-dev
sudo apt install zlib1g-dev
```

(i.e. libcheck plus the development libraries and headers for libfuse and zlib)

Useful information There is a huge amount of documents that you can read about libcheck and libfuse. You do not have to read and understand everything to finish this project, but here are some useful links if you want to learn more.

- Libcheck: https://libcheck.github.io/check/.
- Libfuse: https://github.com/libfuse/libfuse; https://wiki.osdev.org/FUSE.

2. File System Definition

The file system uses a 4KB block size. It is simplified from the classic Unix file system by (a) using full blocks for inodes, and (b) putting all block pointers in the inode. This results in the following differences:

1. There is no need for a separate inode region or inode bitmap – an inode is just another block, marked off in the block bitmap
2. Limited file size – a 4KB inode can hold 1018 32-bit block pointers, for a max file size of about 4MB
3. Disk size – a single 4KB block (block 1) is reserved for the block bitmap; since this holds 32K bits, the biggest disk image is 32K * 4KB = 128MB

Although the file size and disk size limits would be serious problems in practice, they won't be any trouble for the assignment since you'll be dealing with disk sizes of 1MB or less.

File System Format

The disk is divided into blocks of 4096 bytes, and into 3 regions: the superblock, the block bitmap, and file/inode blocks, with the first file/inode block (block 2) always holding the root directory.

```
    +-------+--------+----------+------------------------+
    | super | block  | root dir |     data blocks ...    |
    | block | bitmap |   inode  |                        |
    +-------+--------+----------+------------------------+
  block     0        1          2           3 ...
```

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 fsx_superblock {
    uint32_t magic;             /* 0x30303635 - shows as "5600" in hex dump */
    uint32_t disk_size;         /* in 4096-byte blocks */
    char pad[4088];             /* to make size = 4096 */
};
```

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

Inodes: These are based on the standard Unix-style inode; however they're bigger and have no indirect block pointers. Each inode corresponds to a file or directory; in a sense the inode is that file or directory, which can be uniquely identified by its inode number, which is the same as its block number. The root directory is always found in inode 2; inode 0 is invalid and can be used as a 'null' value.

note: path lookup always begins with inode 2, the root directory, which (see above) is in block 2.

```
struct fs_inode {
    uint16_t uid;      /* file owner */
    uint16_t gid;      /* group */
    uint32_t mode;     /* type + permissions (see below) */
    uint32_t ctime;    /* creation time */
    uint32_t mtime;    /* modification time */
    int32_t  size;     /* size in bytes */
    uint32_t ptrs[FS_BLOCK_SIZE/4 - 5]; /* inode = 4096 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'. (Hint - in C, which you'll be using for testing, octal numbers are indicated by a leading "0", e.g. 0777, so the expression 0777 == 511 is true) The F and D bits correspond to 0100000 and 040000.

side note: Since the demise of 36-bit computers in the early 70s, Unix permissions are about the only thing in the world that octal is used for.

There are a bunch of macros (in `<sys/stat.h>`) you can use for looking at the mode; you can see the official documentation with the command `man inode`, but the important ones are:

- `S_ISREG(m)` - is it (i.e. the inode with mode `m`) a regular file?
- `S_ISDIR(m)` - is it a directory?
- `S_IFMT` - bitmap for inode type bits.

You can get just the inode type with the expression `m & S_IFMT`, and just the permission bits with the expression `m & ~S_IFMT`. (note that `~` is bitwise NOT - i.e. 0s become 1s and 1s become 0s)

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 4096/32 = 128 directory entries in each block. The directory size in the inode is always a multiple of 4096, 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.

for this assignment you can assume directories are always one block in length - the test scripts will never create more than 128 entries in a directory

Storage allocation: Unlike the Linux/Unix file system discussed in lecture, inodes in this file system take up a full block, so there's no need for separate allocation of inodes and blocks. The file system has a single bitmap block, block 1; bit **i** in the bitmap is set if block **i** is in use.

The bits for blocks 0, 1 and 2 will be set to 1 when the file system is created, so you don't have to worry about excluding them when you search for a free block.

side note: Using a single block for the bitmap means that the maximum total file system size is 4K × 8 4KB blocks, or 128MB, which is ridiculously small.

You will need to read this block into a 4KB buffer memory in order to access it (hint: make that buffer a global variable, and read it in your init function), and write it back after you allocate or free a block. (you can do this either at the end of any operations that might allocate/free, or in your allocate/free function itself - I don't care if you write it multiple times)

You're given the following functions to handle the bitmap in memory:

```
int bit_test((void*)map, int i);
void bit_set(unsigned char *map, int i);
void bit_clear(unsigned char *map, int i);
```

3. Implementation advice

1. Subdirectory depth

It's fine to assume a maximum depth for subdirectories, e.g. 10 in the example below.

2. Copying the path

Note that FUSE declares the `path` argument to all your methods as "const char *“, not”char* ", which is really annoying. It means that you need to copy the path before you can split it using `strtok`, typically using the `strdup` function. E.g.:

```
int fs_xyz(const char *c_path, ...)
{
    char *path = strdup(c_path);
    int inum = translate(path);
    free(path);
    ...
}
```

3. Path splitting

The first thing you're probably going to have to do with that path is to split it into components - here's some code to do that, using the `strtok` library function:

```
    #define MAX_PATH_LEN 10
    #define MAX_NAME_LEN 27
    int parse(char *path, char **argv)
    {
        int i;
        for (i = 0; i < MAX_PATH_LEN; i++) {
            if ((argv[i] = strtok(path, "/")) == NULL)
                break;
            if (strlen(argv[i]) > MAX_NAME_LEN)
                argv[i][MAX_NAME_LEN] = 0;        // truncate to 27 characters
            path = NULL;
        }
        return i;
    }
```

4. Factoring

PLEASE factor out the code that you use for translating paths into inodes. Nothing good can come of duplicating the same code in every one of your functions, and you'll find that you've fixed one bug in some of the copies and a different bug in other copies.

I would suggest factoring out a function which translates a path in count/array form (see 6 below) into an inode number. That way you can return an error (negative integer) if it's not found, and accessing the inode itself is just a matter of reading that block number into a `struct fs_inode`. Note that accessing the parent directory is easy using count/array format - it's just (count-1)/array.

5. Efficiency

Don't worry about efficiency. In the simplest implementation, every single operation will read every single directory from the root down to the file being accessed. That's OK.

6. Path translation

Assume you've split your path, and you now have a count and an array of strings:

```
int    pathc;
char **pathv;
```

e.g. for "/home/cs5600" pathc=2, pathv={"home","cs5600"}.

The logic to translate this into the inode number for "/home/cs5600" is:

```
    inum = 2              // root inode
    for i = 0..pathc-1 :
    read inum -> _in
    if _in.mode isn't a directory:
        return -ENOTDIR
    read the directory entries
    search for names[i]
    found:
      inum = dirent.inum
    not found:
      return -ENOENT
```

This will return the inode number of the file/directory corresponding to that path, or -ENOENT/-ENOTDIR if there was an error, with the standard convention that a negative number is an error.

The logic above gives you a function with the signature:

```
int translate(int pathc, char **pathv);
```

7. Translation for mkdir/create/rename/rmdir/unlink

For these functions you need to operate on the containing directory as well as the file/directory indicated by the path. E.g.:

`mkdir /dir1/dir2/parent/leaf`

will not only create a new inode for /dir1/dir2/parent/leaf, but will modify /dir1/dir2/parent to add an entry for it.

If you translate your path into count/array form this is really easy - finding the parent is just `translate(pathc-1, pathv)`, and the name of the leaf is `pathv[pathc-1]`.

(and for e.g. `mkdir` you'll want to verify that the destination **doesn't** exist, i.e. that translate returns an error)

8. `read-img.py`

This utility will print out lots of information about a disk image, which may help figure out what you messed up.

```
$ python read-img.py test.img
blocks used:  0  1  2  21  22  [...]

inodes found:  2  21  55  [...]

inode 2:
  "/" (0,0) 40777 4096
  block 399
    [2] "file.1k" -> 389
    [3] "file.10" -> 268
    [4] "dir-with-long-name" -> 253
    [5] "dir2" -> 213
    [6] "dir3" -> 238
    [7] "file.12k+" -> 327
    [8] "file.8k+" -> 59

inode 389:
  "/file.1k" (500,500) 100666 1000
  blocks:  365
```

The "blocks used" line shows the blocks (including inodes) marked as used in the bitmap; the "inodes found" line shows the inodes found by recursively descending from the root. It's useful for debugging some things; not useful for others.

9. UID/GID, timestamps in `mkdir`, `create`

The "correct" way to get the UID and GID when you're creating a file or directory is this:

```
    struct fuse_context *ctx = fuse_get_context();
    uint16_t uid = ctx->uid;
    uint16_t gid = ctx->gid;
```

Since we're not running FUSE as root, you're only going to see your own UID and GID, so although using `getuid()` and `getgid()` is technically wrong, it's going to give the correct answer in all the test cases.

You can get the timestamp in the proper format with `time(NULL)`. Although the function returns a 64-byte integer, you can assign it to the unsigned 32-bit timestamps in the inode without worrying about overflow for another 85 years or so.

10. `mode` and `mode`

As mentioned in the source code comments and in the next section: The `mode` value passed to `fs_create` will have the inode type bits set - e.g. 0100777 - so you can just put it in the inode mode field. The `mode` value passed to `fs_mkdir` does **not** have the inode type bits set - e.g. 0777 - so you have to OR it with S_IFDIR (giving e.g. 040777).

4. Testing and Debugging Project 3 (and Part 1)

To test the read-only part of your implementation, the makefile will generate a predefined disk image in your directory, named `test.img`; it has known contents and the instructions below give details (e.g. names, sizes, checksums) to verify that your code is reading it properly. You will test your code in two different ways - using the libcheck unit test interface, and running as a FUSE file system.

Testing with libcheck

Your tests will go in the file `unittest-1.c`, which has the #include files and a few other things that you'll need.

The `main` function already has the following parts written for you: - calling `block_init("test.img")`, so `block_read` and `block_write` will work - calling your init method - setting up a test suite, running it, and printing the output

Individual tests are in separate functions that are declared with the START_TEST macro:

```
START_TEST(test_name)
{
    code;
    code;
}
END_TEST
```

For each test you'll need a corresponding call to `suite_add_tcase` in `main`:

```
    tcase_add_test(tc, test_name);
```

In your test functions you'll verify results using libcheck macros. You'll probably want to develop your tests in stages, as any failing tests will cause the entire test run to abort. (If you want to be able to run through all of them you can comment out the line that sets NOFORK mode - that's what I do in the grading tests, but it means that you won't be able to use GDB to debug)

```
ck_abort("message");       /* fail unconditionally */
ck_assert(expr);           /* fail if expression is false */
ck_assert_int_eq(i1, i2);
ck_assert_int_ne/lt/le/ge/gt(i1, i2); /* not eq, less than, etc. */
ck_assert_str_eq("string1", "string2");
ck_assert_ptr_eq(ptr, NULL);
ck_assert_ptr_ne(ptr, NULL); /* assert ptr is NOT null */
```

Constants

The following are defined in the various include files - you'll need most of them.

File type constants - see 'man 2 stat':

- `S_IFMT` - mask for type bits
- `S_IFREG` - regular file
- `S_IFDIR` - directory

A lot of times it's easier to use:

- `S_IFDIR(mode)` - true if it's a directory
- `S_IFREG(mode)` - true if it's a regular file

Error number constants - note that the comments in homework.c will tell you which errors are legal to return from which methods.

- ENOENT - file/dir not found
- EIO - if `block_read` or `block_write` fail
- ENOMEM - out of memory
- ENOTDIR - what it says
- EISDIR - what it says
- EINVAL - invalid parameter (see comments in homework.c)
- EOPNOTSUPP - not finished with the homework yet

Note that you can use `strerror(e)` to get a string representation of an error code. (you'll need to change the error code to a positive number first)

How to test

You'll call the methods in your `fs_ops` structure to get file attributes, read directories, and read files.

You'll need to verify that the data you read from the files is correct. To do that I've provided checksums for the data; once you read it into memory you can calculate your own checksum like this:

```
unsigned cksum = crc32(0, buf, len);
```

then you can check it against the checksums below.

You can write individual tests for each file and directory, but it's a lot easier to loop through a data structure. Here's a suggested way to do that:

```
struct {
    char *path;
    int  len;
    unsigned cksum;  /* UNSIGNED. TESTS WILL FAIL IF IT'S NOT */
    ... other attributes? ...
} table_1[] = {
    {"/this/is/a/file", 120, 1234567},
    {"/another/file", 17, 4567890},
    {NULL}
};

{
    for (int i = 0; table_1[i].path != NULL; i++) {
        do something with table_1[i].path
        check against table_1[i].len, cksum etc.
    }
}
```

The `readdir` method takes a callback function, called `filler` in the parameter list. Here's an example of how to use it; note that the second argument to `readdir` is a `void*` ptr that gets passed as the first argument to the filler. You don't have to worry about `off` or the two parameters to `readdir` that I set to 0 and NULL.

```
int test_filler(void *ptr, const char *name,
            const struct stat *st, off_t off)
{
    struct something *s = ptr;
    printf("file: %s, mode: %o\n", name, st->st_mode);
    return 0;
}

START_TEST(readdir)
{
    struct something s;
    int rv = fs_ops.readdir("/", &s, test_filler, 0, NULL);
    ck_assert(rv >= 0);
}
END_TEST
```

A suggested method for testing `readdir`: create a list of filenames, with a flag for each indicating if it was seen:

```
struct {
    char *name;
    int   seen;
} dir1_table[] = {
    {"file.txt", 0},
    {"dir2", 0},
    {NULL}
};
```

Your filler function can loop through the table looking for a name; if it finds it, it can mark the `seen` flag, and when `readdir` is done you can check that all the `seen` flags have been set. If you want to use the table again (e.g. for multiple tests) you can go back and set all the `seen` flags to zero. Oh, and your filler should probably flag an error if it doesn't find a name, or if the name is already marked "seen".

Contents of test.img

First, the values returned by `getattr` in `struct stat`, for all files and directories:

```
# path uid gid mode size ctime mtime
"/", 0, 0, 040777, 4096, 1565283152, 1565283167
"/file.1k", 500, 500, 0100666, 1000, 1565283152, 1565283152
"/file.10", 500, 500, 0100666, 10, 1565283152, 1565283167
"/dir-with-long-name", 0, 0, 040777, 4096, 1565283152, 1565283167
"/dir-with-long-name/file.12k+", 0, 500, 0100666, 12289, 1565283152, 1565283167
"/dir2", 500, 500, 040777, 8192, 1565283152, 1565283167
"/dir2/twenty-seven-byte-file-name", 500, 500, 0100666, 1000, 1565283152, 1565283167
"/dir2/file.4k+", 500, 500, 0100777, 4098, 1565283152, 1565283167
"/dir3", 0, 500, 040777, 4096, 1565283152, 1565283167
"/dir3/subdir", 0, 500, 040777, 4096, 1565283152, 1565283167
"/dir3/subdir/file.4k-", 500, 500, 0100666, 4095, 1565283152, 1565283167
"/dir3/subdir/file.8k-", 500, 500, 0100666, 8190, 1565283152, 1565283167
"/dir3/subdir/file.12k", 500, 500, 0100666, 12288, 1565283152, 1565283167
"/dir3/file.12k-", 0, 500, 0100777, 12287, 1565283152, 1565283167
"/file.8k+", 500, 500, 0100666, 8195, 1565283152, 1565283167
```

File lengths and checksums:

```
# checksum length file
1786485602, 1000, "/file.1k"
855202508, 10, "/file.10"
4101348955, 12289, "/dir-with-long-name/file.12k+"
2575367502, 1000, "/dir2/twenty-seven-byte-file-name"
799580753, 4098, "/dir2/file.4k+"
4220582896, 4095, "/dir3/subdir/file.4k-"
4090922556, 8190, "/dir3/subdir/file.8k-"
3243963207, 12288, "/dir3/subdir/file.12k"
2954788945, 12287, "/dir3/file.12k-"
2112223143, 8195, "/file.8k+"
```

Directory contents (for testing readdir):

```
"/" : "dir2", "dir3", "dir-with-long-name", "file.10",
      "file.1k", "file.8k+"
"/dir2" : "twenty-seven-byte-file-name", "file.4k+"
"/dir3" : "subdir", "file.12k-"
"/dir3/subdir" : "file.4k-", "file.8k-", "file.12k"
"/dir-with-long-name" : "file.12k+"
```

Suggested tests (i.e. the grading tests)

`fs_getattr` - for each of the files and directories, call `getattr` and verify the results against values from the table above.

`fs_getattr` - path translation errors. Here's a list to test, giving error code and path: * ENOENT - "/not-a-file" * ENOTDIR - "/file.1k/file.0" * ENOENT on a middle part of the path - "/not-a-dir/file.0" * ENOENT in a subdirectory "/dir2/not-a-file"

`fs_readdir` - check that calling readdir on each directory in the table above returns the proper set of entries.

`fs_readdir` errors - call readdir on a file that exists, and on a path that doesn't exist, verify you get ENOTDIR and ENOENT respectively.

`fs_read` - single big read - The simplest test is to read the entire file in a single operation and make sure that it matches the correct value.

I would suggest using a buffer bigger than the largest file and passing the size of that buffer to `fs_read` - that way you can find out if `fs_read` sometimes returns too much data.

`fs_read` - multiple small reads - write a function to read a file N bytes at a time, and test that you can read each file in different sized chunks and that you get the right result. Note that there's no concept of a current position in the FUSE interface - you have to use the offset parameter. (e.g read (len=17,offset=0), then (len=17,offset=17), etc.)

(I'll test your code with N=17, 100, 1000, 1024, 1970, and 3000)

`fs_statvfs` - run 'man statvfs' to see a description of the structure. The values for the test image should be: * `f_bsize` - 4096 (block size, bytes) * `f_blocks` - 1024 (total number of blocks) * `f_bfree` - 731 (free blocks) * `f_namemax` - 27

Methods that modify the disk image Both `chmod` and `rename` will (obviously) modify the disk image. Be sure to regenerate the test image each time you run your tests.

Hint - you'll never forget to do that if you put the statement `system("python gen-disk.py -q disk1.in test.img")'` at the beginning of your `main` function in `unittest-1.c`.

`fs_chmod` - change permissions for a file, check that (a) it's still a file, and (b) it has the new permissions. Do the same for a directory. (note that `chmod` should only change the bottom 9 bits of the mode word)

`fs_rename` - try renaming a file and then reading it, renaming a directory and then reading a file from the directory.

Running as a FUSE file system

If you build and run the `hw3fuse` executable you can run it with the following command:

```
./hw3fuse -image test.img [dir]
```

and (assuming it doesn't crash) it will mount the file system in test.img on top of the specified directory and run in the background, letting us see the contents:

```
hw3$ mkdir dir
hw3$ ./hw3fuse -image test.img dir
hw3$ ls dir
dir2  dir3  dir-with-long-name  file.10  file.1k  file.8k+
hw3$ ls -l dir
total 6
drwxrwxrwx 1  500  500  8192 Aug  8 12:52 dir2
drwxrwxrwx 1 root  500  4096 Aug  8 12:52 dir3
drwxrwxrwx 1 root root  4096 Aug  8 12:52 dir-with-long-name
-rw-rw-rw- 1  500  500    10 Aug  8 12:52 file.10
-rw-rw-rw- 1  500  500  1000 Aug  8 12:52 file.1k
-rw-rw-rw- 1  500  500  8195 Aug  8 12:52 file.8k+
hw3$ cd dir
hw3/dir$ ls -l dir2
total 2
-rwxrwxrwx 1 500 500 4098 Aug  8 12:52 file.4k+
-rw-rw-rw- 1 500 500 1000 Aug  8 12:52 twenty-seven-byte-file-name
hw3/dir$
```

To unmount the directory:

```
hw3$ fusermount -u dir
hw3$
```

Note that you have to unmount the directory even (or especially) if your program crashes.

`Transport endpoint is not connected` This means your program crashed and you didn't use `fusermount -u` to clean up after it.

To use GDB in FUSE mode you'll want to specify two additional flags that get interpreted by the FUSE library: `-s` specifies single-threaded mode, and `-d` specifies debug mode so it runs in the foreground.

```
gdb --args ./hw3fuse -s -d -image test.img dir
```

5. Testing part 2

First, make sure you create a new disk image each time you test

When you run buggy code and write to your disk, it's going to mess up the disk image. When you fix the bug, your code might not work if you don't replace the corrupted disk image. You'll be unhappy. Don't do that.

You can make an empty disk named `test2.img` with the following command:

```
$ python gen-disk.py -q disk2.in test2.img
```

(because I haven't written `mkfs` yet…)

To test your write logic (including mkdir, create, etc.) you need to be able to trust your `readdir` and `read` implementations, so that you can verify the results. That's the primary reason why the assignment is explicitly split into two parts.

`fuse_getcontext` mocking

We're using a test strategy called "mocking" to deal with the `fuse_getcontext` call - we provide a fake version of the function in our test code, and in each test we can fill in any information we want it to return.

create/mkdir/unlink/rmdir

WARNING: The `mode` parameter passed to `create` must have the inode type bits set properly for a file - e.g. 0100777 (or S_IFREG | 0777) The `mode` parameter passed to `mkdir` **does not have those bit set** - e.g. 0777.

Non-error cases - these are fairly simple:

- create multiple subdirectories (with `mkdir`) in a directory, verify that they show up in `readdir`
- delete them with `rmdir`, verify they don't show up anymore
- create multiple files in a directory (with `create`), verify that they show up in `readdir`
- delete them (`unlink`), verify they don't show up anymore

Now run these tests (a) in the root directory (i.e., the mount point), (b) in a subdirectory (e.g. "/dir1") and (c) in a sub-sub-directory (e.g. "/dir1/dir2")

Once you've written the tests to run in the root directory, it should be simple (either a loop or copy and paste) to add cases (b) and (c).

Once you've implemented `write`, add an additional set of unlink tests where you write data into the files first. You might want to use `statvfs` to verify that unlink freed the blocks afterwards. (the grading tests will do that)

Error cases - you should test the following

create: - bad path /a/b/c - b doesn't exist (should return -ENOENT) - bad path /a/b/c - b isn't directory (ENOTDIR) - bad path /a/b/c - c exists, is file (EEXIST) - bad path /a/b/c - c exists, is directory (EEXIST) - too-long name (more than 27 characters)

For the long name case, you're allowed to either return -EINVAL or truncate the name.

mkdir: - bad path /a/b/c - b doesn't exist (ENOENT) - bad path /a/b/c - b isn't directory (ENOTDIR) - bad path /a/b/c - c exists, is file (EEXIST) - bad path /a/b/c - c exists, is directory (EEXIST) - too-long name

unlink: - bad path /a/b/c - b doesn't exist (ENOENT) - bad path /a/b/c - b isn't directory (ENOTDIR) - bad path /a/b/c - c doesn't exist (ENOENT) - bad path /a/b/c - c is directory (EISDIR)

rmdir: - bad path /a/b/c - b doesn't exist (ENOENT) - bad path /a/b/c - b isn't directory (ENOTDIR) - bad path /a/b/c - c doesn't exist (ENOENT) - bad path /a/b/c - c is file (ENOTDIR) - directory not empty (ENOTEMPTY)

write - test data

To test write you'll need test data. For debugging it you'll want to be able to figure out what went wrong, so I'm going to suggest you put a repeating pattern into your data - here's an example, where I want to write 4000 bytes:

```
char *ptr, *buf = malloc(4010); // allocate a bit extra
int i;
for (i=0, ptr = buf; ptr < buf+4000; i++)
    ptr += sprintf(ptr, "%d ", i);
fs_ops.write("/path", buf, 4000, 0, NULL);  // 4000 bytes, offset=0
```

So you get something that looks like:

```
(gdb) l
5   {
6       char *ptr, *buf = malloc(4000);
7       int i;
8       for (i=0, ptr = buf; ptr < buf+4000; i++)
9           ptr += sprintf(ptr, "%d ", i);
10      printf("breakpoint here\n");
11  }
(gdb) p buf
$1 = 0x5555555592a0 "0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
(gdb)
```

Two other stragegies: - fill in each byte with a random number using `random()` - fill in bytes with 0, 1, 2, …

The first strategy is great for testing someone else's code (I use it for the grading tests) but not so good for debugging. The second approach isn't so great, because you're going to be making mistakes with 4KB blocks of data, so using a pattern that wraps around every 256 bytes might not show the problems.

write - tests

append tests You should create files of various sizes (at a minimum: <1 block, 1 block, <2 blocks, 2 blocks, <3 blocks, 3 blocks), writing to them with the same buffer sizes that you used in your read tests. Note that (like read) you'll have to calculate the offset yourself - there's no file pointer in the FUSE interface.

You'll have to read the data back into a different buffer, then you can verify that the original data and the data you read back are the same. You can use checksums to do this, or the `memcmp` function, or something else if you want.

Now unlink them, and use `statvfs` to verify that the number of free blocks when you're done is the same as the number of free blocks before you created the files and wrote to them.

overwrite tests For this you'll need two buffers, with different contents. (hint - if you use the code above to generate your data, just start at a different number)

Write the first buffer to the file, then write the second buffer, but starting over at offset 0. Now read the file back and verify that the contents match the second buffer.

Oh, I assume you're checking the return value from `write` and other FUSE functions each time you use them, and verifying that they =0 the times they should succeed, right?

truncate

Verify that you can truncate files of the same set of sizes that you use in the write tests. Use `statvfs` to verify that the blocks have been freed.

6. What to submit

You should submit all files related to the project in a zip file. For this project, write all your code within homework.c, unittest-1.c, and unittest-2.c files and you do not have to submit a README.txt file. Your unittest-X.c file should include at least one test case for each fs_XXX function that you wrote. Unlike previous projects, this project will be graded mostly based on running/testing your code.