Assignment 4: The Very Simple File System

In this assignment, you implement a version (Section 2) of the Very Simple File System (VSFS) from the OSTEP textbook and our lectures. This assignment uses FUSE, which allows you to implement a file system in user space by implementing the callback functions that the `libfuse` library will call. The Tutorial 7 exercise should give you some practice using FUSE. **We highly recommend starting there.**

If you would like to learn more about FUSE:

- GitHub repository
- FUSE API header file for the version we're using
- FUSE wiki

Contents

1 Logistics

Before starting to program, we highly recommend reading the entirety of this handout. This section describes how to acquire the starter files, compile code, and submit the assignment.

1.1 Setting up your repository

Your assignment is downloaded from, and submitted to, MarkUs using `git`. Login to MarkUs and navigate to CSC369. From there, navigate to 'a4', where you can see (among other things) the URL of the repository. Before cloning the repository, click 'Add Starter Files to Repository'. Now you can clone the repository.

1.2 Working in your repository

Once you have cloned your repository you should find the starter files in the 'a4' subdirectory. Start by reading the `README.md` in that subdirectory, which includes instructions on how to compile everything. The majority of your (graded) work should be done in `a4/src/mkfs.c` and `a4/src/vsfs.c`, which correspond to two different executables. Scripts are provided to run these executables in the `a4/scripts` directory (Sections 4.1 and 5.4). In addition, you can (and are encouraged to) add unit tests using Python in `a4/tests` (Section 5.4).

1.3 Submitting changes to your repository

You can find the files you have modified by running the `git status` command. You can commit modified files to your local repository. For example, assuming you are in the `a4/src` directory and have saved changes to `mkfs.c`:

```
git add mkfs.c
git commit -m "My useful commit message"
```

We suggest committing your changes frequently by rerunning the commands above (with different meaningful messages to the commit command), so that you can go back to see the changes you have made over time. Once you have tested your code, and committed it locally (check that by running git status), you can `git push` it back to MarkUs. You should also do this frequently. We collect and grade the last version pushed to MarkUs before the due date, up to the maximum late penalty. Please see the Syllabus and MarkUs for more details on the late penalty.

2 Assumptions, limits, and implementation details

We will not be testing your code under extreme circumstances so don't get carried away by thinking about corner cases. However, we do expect you to properly handle "out of space" conditions in your code. Any operation that cannot be completed because there are not enough free blocks or inodes must be cleanly aborted - no blocks or inodes can "leak" in the process. The simplest way to ensure this is to check that there is enough space to complete the operation before modifying any file system metadata. The formatting program (Section 4) must also check that the image file is large enough to accommodate the requested number of inodes.

Here are some hints and assumptions:

1. The maximum number of inodes in the system is a parameter to `mkfs.vsfs`, the image size is also known to it, and the block size is `VSFS_BLOCK_SIZE` (4096 bytes - declared in `vsfs.h`). Many parameters of your file system can be computed from these three values.

2. We will not test your code on an image smaller than 64 KiB (16 blocks) with 4 inodes.

   You should be able to fit the root directory and a non-empty file in an image of this size and configuration. You shouldn't pre-allocate additional space for metadata (beyond the fixed metadata definced for VSFS, the space needed to store the inode table and the root directory) in your `mkfs.vsfs` implementation. **Indirect blocks should only be allocated on demand**, when a file or directory grows large enough to need it.

3. The maximum path component length is `VSFS_NAME_MAX` (252 bytes including the null terminator).

   This value is chosen to fit the directory entry structure into 256 bytes (see `vsfs.h`). Names stored in directory entries are null-terminated strings so that you can use standard C string functions on them.

4. The maximum full path length is `_POSIX_PATH_MAX` (4096 bytes including the null terminator).

   This allows you to use fixed-size buffers for operations like splitting a path into a directory name and a file name.

5. The maximum file size is based on the number of direct block pointers in an inode (`VSFS_NUM_DIRECT`) and the number of block pointers in an indirect block (`VSFS_BLOCK_SIZE / sizeof(vsfs_blk_t)`).

6. The number of directory entries is limited by the maximum number of directory entry data blocks (same as the limit on file blocks).

7. The number of blocks in your file system is limited by the number of bits in a single VSFS block, since we use only 1 block for the data bitmap.

8. You can assume that read and write operations are performed one block at a time.

Each `read` and `write` call your file system receives will only cover a range within a single block. **Note**: this does not apply to `truncate` - a single call needs to be able to extend or shrink a file by an arbitrary number of blocks.

Sample disk configurations that must work include:

- 64KiB size and 4 inodes
- 64KiB size and 16 inodes
- 1MiB size and 64 inodes
- 128MiB size and 512 inodes

Other implementation notes:

- Although the "." and ".." directory entries can be manually listed by the `vsfs_readdir` callback (as in the starter code), you should create actual entries for these when you initialize the root directory in `mkfs`.
- The only timestamp you need to store for each file and directory is mtime (modification time) - **you don't need to store atime and ctime**. You can use the `touch` command to set the modification timestamp of a file or directory to current time.
- Any data and metadata blocks (other than the fixed metadata) should only be allocated on demand.
- Read and write I/O should be performed by reading/writing the virtual memory where the disk image is mmap'd. It should NOT be performed byte-by-byte (which is very inefficient); use `memcpy`.
- Your implementation shouldn't use any floating point arithmetic. See the helper functions in `util.h` - if you need other, similar, functions (like floor), they can also be easily implemented using integer arithmetic.

3 Background

Unlike the passthrough file system of the tutorial exercise, your VSFS file system will operate on a disk image file. A disk image is an ordinary file that holds the content of a disk partition or storage device. We covered disk images in Lecture 18. In this assignment you work on two executables: `mkfs.vsfs` and `vsfs`. The `mkfs.vsfs` executable is responsible for formatting a disk image (Section 4). The `vsfs` executable mounts a disk image to a mount point (Section 5), allowing the user to perform operations (like `ls`).

3.1 Disk images

To allow you to test file system operations independently of `mkfs`, we have provided some simple VSFS-formatted disk images at `/u/csc369h/winter/pub/a4`:

1. vsfs-empty.disk - Small, empty file system (64 inodes, 1 MB size). Contains just root directory with '.' and '..' entries.

2. vsfs-1file.disk - Small file system (64 inodes, 1 MB size) containing a single small file (only 1 data block) in the root directory.

3. vsfs-3file.disk - Medium file system (128 inodes, 16 MB size) containing 3 files (small - only direct blocks, medium - some indirect blocks, and maximum VSFS file size).

4. vsfs-many.disk - Small file system (256 inodes, 2 MB size) containing lots of small files (root directory inode uses indirect block pointer).

5. vsfs-maxfs.disk - Maximum size VSFS file system (512 inodes, 128 MB size). Contains just root directory with '.' and '..' entries.

You will need to make your own copies of these disk images to use them, since you will need to be able to write to them.

3.2 Consistency checkers

The name `fsck` comes from the common tool for checking the consistency of file systems in Unix-like operating systems. Available on the `teach.cs` machines are two executables for checking the consistency of images. The first executable, `fsck.mkfs`, is useful in ensuring your `mkfs` implementation correctly formats the disk. The second executable, `fsck.vsfs`, is useful in ensuring that your code that was called during different file system operations has not corrupted the disk somehow.

4 Formatting

This part of the assignment does not need FUSE at all. You need to write part of the `mkfs.c` program. To better understand how to complete this part of the assignment, make sure you have carefully read Chapter 40. You should also refer to Section 2 for simplifying assumptions you can make.

You will find it helpful to read the code to see how we access parts of the file system after using `mmap` to map the entire disk image into the process virtual address space. Our goal is to support the following use case:

```
truncate -s <size> <image file>
./mkfs.vsfs -i <number of inodes> <image file>
```

The truncate command will create a file (if it doesn't exist) and will set its size. The `mkfs.vsfs` executable (i.e., your executable) formats that file into a vsfs file system (i.e., a 'disk' or a 'disk image'). Once complete, we can check the consistency of file using `fsck.mkfs`.

4.1 Automated running and testing

The process of creating a file, formatting it, and then checking its consistency has been automated for you. The script `a4/scripts/run_mkfs.sh` does each step, redirecting the output of these executables to files. For example, from the script:

Excerpt from run_mkfs.sh

```
test_mkfs "mkfs_small" 1048576 64          # 1 MB
test_mkfs "mkfs_medium" 16777216 128       # 16 MB
```

Consider adding more calls to `test_mkfs` in this script, varying the file size and number of inodes. The disk images already provided to you (Section 3) may help give you some ideas. You can find the `test_mkfs` bash function in `util.sh`. To run the script itself, you must provide a single argument: the path to your build directory where you have compiled your project. For example, assuming you are in the `a4` directory and have built your project in `a4/cmake-build-debug`:

```
csc369$ ./scripts/run_mkfs.sh cmake-build-debug
+ exec
Creating the image file cmake-build-debug/test_mkfs/mkfs_small.disk
Making a file system with 64 inodes.
Checking consistency.
Done. See cmake-build-debug/test_mkfs for all redirected output files.
Creating the image file cmake-build-debug/test_mkfs/mkfs_medium.disk
Making a file system with 128 inodes.
Checking consistency.
Done. See cmake-build-debug/test_mkfs for all redirected output files.
```

5 Mounting

Once you have a formatted vsfs disk image (one of ours, or your own) the next step is to mount your file system. Our goal is to support the following use case: `./vsfs <image file> <mount point> -f`

The image file is the disk image formatted by mkfs.vsfs. Not only does `vsfs` mount the disk image into the local file system, it also sets up callbacks and then calls `fuse_main` so that FUSE can do its work. That is, after the file system is mounted, you can access it using standard tools (ls, cp, rm, etc.). To unmount the file system, run: `fusermount -u <mount point>` You should be able to unmount the file system after any sequence of operations, such that when it is mounted again, it has the same contents.

5.1 Understanding the starter code

You should read the starter code to understand how it fits together, and find useful helper functions. Here is a brief summary of the relevant files for mounting:

- `vsfs.h` - contains the data structure definitions and constants needed for the file system. You may add other definitions or constants that you find useful, but you should not change the file system metadata. That is, do not add or modify fields in the superblock, inode, or direntry structures and do not change the existing definitions.

- `vsfs.c` - contains the program used to mount your file system. This includes the callback functions that will implement the underlying file system operations. Each function that you will implement is preceded by detailed comments and has a "TODO" in it. Please read this file carefully.

  Note: It is very important to return the correct error codes (or 0 on success) from all the FUSE callback functions, according to the "Errors" section in the comment above the function. The FUSE library, the kernel, and the user-space tools used to access the file system all rely on these return codes for correctness of operation.

  Note: You will see many lines like (void)fs;. Their purpose is to prevent the compiler from warning about unused variables. You should delete these lines as you make use of the variables.

- `fs_ctx.h` and `fs_ctx.c` - The `fs_ctx` struct contains runtime state of your mounted file system. Any time you think you need a global variable, it should go in this struct instead. We have cached some useful global state in this structure already (e.g. pointers to superblock, bitmaps, and inode table), but you may find there is additional state that you want to add, instead of recomputing it on every operation.

- `map.h` and `map.c` - contain the `map_file` function used by vsfs and mkfs.vsfs to map the image file into memory and determine its size. You should not need to change anything here, or make any additional calls to the `map_file` function beyond what is in the starter code.

- `options.h` and `options.c` - contain the code to parse command line arguments for the vsfs program. You should not need to change anything here.

- `util.h` - contains some handy functions.
- `bitmap.h` and `bitmap.c` - contain code to initialize bitmaps, and to allocate or free items tracked by the bitmaps. You will use these to allocate and free inodes and data blocks, so make sure you read the functions and understand how to use them. You may notice that the `bitmap_alloc` function can be slow, since it always starts the search for a 0 bit from the start of the bitmap. You are free to improve on this if you wish, but you do not need to.

5.2 Recommended progression

There are a lot of places where you can start for mounting in terms of writing code. However, there is an incremental order of operations that makes it much easier to test things. To help, we have already implemented `vsfs_statfs` because it is very useful for testing. Next, we recommend you follow this order:

1. Implement `vsfs_getattr`. You have probably seen from the tutorial exercise that FUSE calls `getattr` a lot. Implementing this function is the key to the rest of the operations.

   You will want to write a helper function that takes a path and returns a pointer to the inode (or the inode number) for the last component in the path. Remember (Section 2) that you only need to handle paths that are of the form “/” or “/somefile” - all paths are absolute and there are no subdirectories in our vsfs file systems.

2. Implement `vsfs_readdir`. Once done correctly, you can run `ls -la` on the root directory when the root directory entries fit within a single data block. You should be able to mount `vsfs-empty.disk`, `vsfs-maxfs.disk`, `vsfs-1file.disk` and `vsfs-3file.disk` and list their root directories on completion of this step.

3. Add the ability to create empty files by implementing `vsfs_create`. On completion of this step, you should be able to mount `vsfs-empty.disk` and use touch to create a new empty file. The new file should be visible and the mode and timestamps should be appropriate when you `ls -l` on the root directory.

4. Add the ability to write to, and read from, small files. First focus on the case where the data is stored in a single data block. Then the case where the data can be stored using only the direct block pointers in the inode. Implement `vsfs_truncate` first, then `vsfs_write` and `vsfs_read`.

5. Add the ability to remove small files (where the file data uses only the direct block pointer in the inode).

6. Enhance your implementation of `vsfs_readdir` to list large directories, where the directory inode’s indirect block is needed to read all of the directory data blocks. You should be able to mount `vsfs-many.disk` and list its root directory on completion of this step.

7. Enhance your implementations of `vsfs_truncate`, `vsfs_write`, `vsfs_read`, and `vsfs_unlink` to support large files, where the indirect block in the file’s inode is used to locate some of the file’s data blocks.

While making changes and supporting more operations, here are some tips:

- Remember to update fields in the superblock (e.g. `free_inodes`, `free_blocks`) as you operate on the file system.

- Check that there is enough space **before** making any changes to the file system. This will save you from having to roll back changes if you discover that an operation cannot be completed due to lack of space.

- Comment your code well. It will help you keep track of what is implemented and your understanding of how things work. Refactor your code during development (not after) and keep your functions short and well-structured.

5.3 Testing and debugging recommendations

You can use standard Unix tools to manipulate directories and files in a mounted vsfs image in order to test your implementation. System call tracing with `strace` can help understand what syscalls they invoke to access the file system. You can, in general, use the behaviour of the host file system (ext4) as a reference - your `vsfs` should have the same observable behaviour for operations that `vsfs` needs to support.

You will find it useful to run `vsfs` under `gdb`: `gdb --args ./vsfs <image file> <mount point> -d` You can then run file system operations in a separate terminal window. You can set breakpoints at the start of your FUSE callback functions (e.g. `break vsfs_getattr`) to help you understand what callbacks are invoked when you execute a file system operation (e.g. `ls`), in what order, and with what arguments. The debugger is also helpful in investigating crashes (e.g., segfaults) and stepping through the execution of the callback functions so that you can check your the state of the filesystem as the operations execute. Off-by-one errors are common but can be catastrophic when they lead to accessing the wrong block of file system metadata.

You might also find it useful to view the binary contents of your vsfs image files using `xxd`. See `man 1 xxd` for documentation.

To avoid errors when mounting the file system, make sure that the mount point is not in use (e.g., by a previous vsfs mount that didn't finish cleanly). If `fusermount` fails to unmount because the mount point directory is "busy", you can use the `lsof` command (see `man lsof`) to identify the process that keeps it open. One common error message you might see when running operations on the mounted file system is "transport endpoint is not connected". This error usually means that the file system is still mounted, but the `vsfs` program has terminated (e.g. crashed). In this case you need to manually unmount it with `fusermount`.

One of the most common errors you might see at the early stages of the implementation is `ls -la` reporting an "I/O error" and displaying "???" entries. This error usually means that your `getattr` callback returns invalid data in the stat structure and/or an invalid return value.

You can also write your own C programs that invoke relevant syscalls directly, or use Python unit tests (Section 5.4). In order to test reads and writes at an offset, you can either use the `tail` command (its `-c` option; see `man 1 tail`). Or write your own C programs that use `pread` and `pwrite`.

5.4 Automated running and testing

The process of mounting an image, running operations, and unmounting (and, finally, checking the consistency of the image) has been automated for you in two ways. The script `a4/scripts/run_vsfs.sh` does each step, redirecting the output of these executables to files. For example, from the script:

Excerpt from run_vsfs.sh

```
test_vsfs "vsfs-empty" "$CSC369_A4_DISK_DIR/vsfs-empty.disk" 64 "$TESTS_DIR"
test_vsfs "vsfs-1file" "$CSC369_A4_DISK_DIR/vsfs-1file.disk" 64 "$TESTS_DIR"
```

However, the script does not know which operations to try on the file system. Instead, it runs `pytest` on the `a4/tests` directory, providing some useful command line arguments. You can find the `test_vsfs` bash function in `scripts/util.sh` (find the line that begins with `python3`). Inspect `a4/tests/conftest.py` for a better understanding of what `pytest` is provided by the `run_vsfs.sh` script. Then take a look at `test_statvfs.py`, which includes test cases for validating the correctness of the `statfs` operation.

To run the script itself, you must provide two arguments: the path to your build directory and the tests directory. For example, assuming you are in the `a4` directory and have built your project in `a4/cmake-build-debug`:

```
csc369$ ./scripts/run_vsfs.sh cmake-build-debug/ tests/
+ exec
Starting vsfs-empty test
Mounting /u/csc369h/winter/pub/a4/vsfs-empty.disk at /tmp/mbadr
Running pytest tests in tests/.
Unmounting /tmp/mbadr
Checking consistency.
Done. See cmake-build-debug//test_vsfs for all redirected output files.
...
```

There are several other files in `a4/tests` for testing different operations. However, they are (mostly) empty skeletons. You can add tests to the existing files, and we have provided some hints for you for some operations. If you are adding a new test to a file, remember that the function must have the prefix `test_`. If you are adding a new file to test a different operation, remember that the file must also have the prefix `test_`.

Please note that your test cases are not assessed. However, reading, understanding, and writing test cases is incredibly important for this assignment. As a result, we encourage you to **share your tests on Piazza.** To do so, create a new Piazza post with a subject that contains the name of the file (e.g., 'marsbar's test_readdir.py'). Then, share the contents of that file using the 'Markdown editor' and 'backticks' (don't use an attachment):

~~~
```python
# paste your code here
```
~~~

Other students (and your instructor) can now see your test cases, reason about their validity, and make suggestions. If you make changes that you want to share, **edit your original post**. Students are free to use each others' test cases. Outside of Python test cases, however, no sharing of files is allowed. If you have a question or doubt about academic integrity, please ask your instructor.