# Assignment 1b: File System Implementation

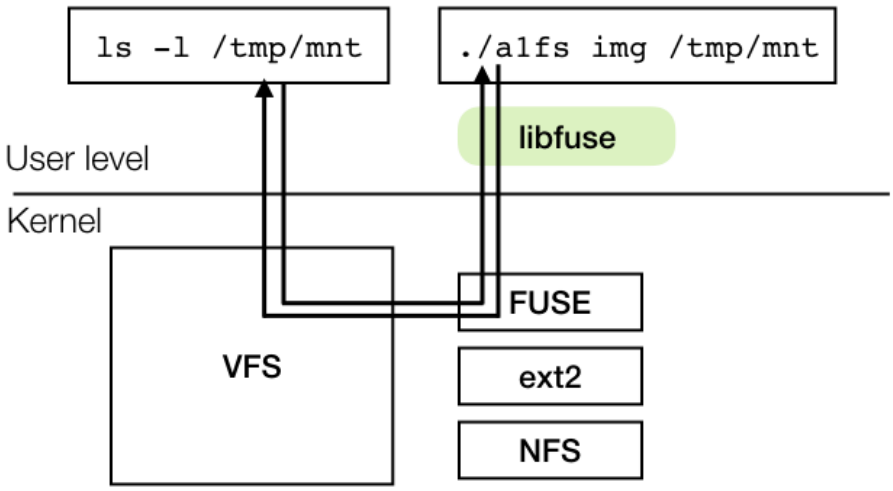
**Due** Oct 22, 2021 by 10p.m. **Points** 10

Edit: I wrote a couple of little programs that will help you test read and write on your file system. I wrote them to try stuff out, so I have not extensively tested them.

- **read_test.c (https://q.utoronto.ca/courses/234190/files/16652484/download?download_frd=1)**
- **write_test.c (https://q.utoronto.ca/courses/234190/files/16652485/download?download_frd=1)**

## Introduction

You will want to get started working on your file system implementation even before you get feedback on your proposal. We will be using FUSE to interact with your file system. FUSE allows you to implement a file system in user space by implementing the callback functions that the `libfuse` library will call. The diagram below shows how the FUSE kernel module receives file system calls in the kernel and passes them to your file system program. We will also be providing you will additional starter code.




This diagram shows the layers involved in using your file system code with FUSE. In the top right it shows that a1fs has been executed to mount your file system at `/tmp/mnt`. Now when you run `ls -l /tmp/mnt` the `ls` program will execute the system calls to list the contents of the directory. These system calls are executed in the kernel which recognizes that they need to be handled by the FUSE file system. FUSE executed the callbacks that you wrote in `a1fs`. For `ls`, this will include `a1fs_getattr()` and `a1fs_readdir()`

## Using FUSE

The tutorial notes will include information on getting started with FUSE, and check Piazza for tips. There are two recommended ways to work with FUSE for the assignment. Both of them will require some facility with the command-line file system operations. To be able to use FUSE, it must be running in your kernel, so this means that it is not feasible to run the code for this tutorial on your Mac OS, Windows, WSL on Windows. These operating systems have FUSE installed, but they have not turned on the options that allow you to implement your own

file system. If you do figure out how to do it on WSL or Mac OS, please post the instructors to Piazza.

### A. Using `csc369.teach.cs.toronto.edu`

Our system administrators have set up a server just for CSC369 with FUSE installed. To log into this server, you must first log into `teach.cs.toronto.edu`, and from there ssh into `csc369.teach.cs.toronto.edu`.

For example, from a terminal on my machine, I run:

```
ssh USERID@teach.cs.toronto.edu
```

(Here and below `USERID` must be replaced with your own user id.)

Then in that terminal, run

```
ssh csc369.teach.cs.toronto.edu
```

to log into the csc369 server. You still have access to all of your files on the teach.cs machines, so you can clone your repository for this exercise.

You will need to mount your FUSE file system in a sub-directory of `/tmp`, e.g. `./a1fs img /tmp/USERID`, and not in your home directory. FUSE needs full permissions on the path to the mount point (which is not the case for your home directory), otherwise you can wind up with directories that you can't remove and can't use.

To use `gdb`, you must start `a1fs` under `gdb`. You won't be able to attach `gdb` to a running `a1fs` process.

### B. Using a VM on your own machine

You can run FUSE using an Ubuntu virtual machine. Here are our recommendations for the setup:

- **VirtualBox** (https://www.virtualbox.org) is a free open-source hypervisor that allows you to run a guest OS on top of your machine. It is pretty easy to install.
- Now install Ubuntu 18.04 as a VM in VirtualBox. I used an image from **osboxes.org** (https://www.osboxes.org/ubuntu/), but you may find others. You can find **tutorials** (https://www.wikihow.com/Install-Ubuntu-on-VirtualBox) that explain the process.
- Make sure your FUSE version is 2.9.* (run `fusermount --version`).

- Install additional packages:
  - `sudo apt-get install libfuse-dev pkg-config`
- To attach `gdb` to a running process you will need to change a security setting on Linux. You can do that using the following commands:
  - `sudo bash`
  - `echo 0 > /proc/sys/kernel/yama/ptrace_scope`
- You may want to increase the screen resolution for the VM. You can do this by selecting the "Show Applications" button in the bottom left corner and then going to Settings->Devices->Displays and changing the resolution.

## Running the programs

To create a disk image you will run the following commands:

```
truncate -s <size> <image file>
./mkfs.a1fs -i <number of inodes> <image file>
```

`truncate` will create the image file if it doesn't exist and will set its size. `mkfs.a1fs` will format it into your a1fs file system.

The next step is to mount your file system. You will need to choose a location in the system's file tree to "graft" your files.

```
./a1fs <image file> <mount point>
```

The image file is the disk image formatted by `mkfs.a1fs`. Not only does `a1fs` mount the disk image into the local file system, but it also sets up callbacks and then calls `fuse_main()` so that FUSE can do its work. Both `a1fs` and `mkfs.a1fs` have additional options - run them with `-h` to see their descriptions.

After the file system is mounted, you can access it using standard tools. To unmount the file system, run:

```
fusermount -u <mount point>
```

Note that 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.

## Understanding the code

First read through all of the starter code to understand how it fits together, and which files contain helper functions that will be useful in your implementation.

- `mkfs.c` - contains the program to format your disk image. You need to write part of this program.
- `a1fs.h` - contains the data structure definitions and constants needed for the file system. You wrote this for your proposal, but you may find you need to make changes as you work on the implementation. Also, we've added some useful documentation comments since the a1a version of this file, so please make sure to use the updated version.
- `a1fs.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. You will find it useful to cache some global state (e.g. pointers to bitmaps, inode table, root inode, etc.) in this structure instead of recomputing them for every operation.
- `map.h` and `map.c` - contain the `map_file()` function used by `a1fs` and `mkfs.a1fs` to map the image file into memory and determine its size.
- `options.h` and `options.c` - contain the code to parse command line arguments for the `a1fs` program.
- `util.h` - contains some handy functions.

You are welcome to put some of the helper functions in separate files instead of keeping everything in `a1fs.c`. Make sure to update the `Makefile` to compile those files and add/commit/push them to your git repository.

## Recommended progression of your work

You should tackle this project in stages so that you can be confident that each piece works before moving on to the next step.

Step 1: Write enough of `mkfs.a1fs` so that you can mount the file system and see the root directory.

Step 2: Implement `a1fs_statfs()` and test it by running `stat -f` on the root directory to check that the superblock is initialized correctly by `mkfs`.

Step 3: Write `a1fs_getattr()` and `a1fs_readdir()` so that you can run `ls -la` on the root directory.

Step 4: Implement the code to create and traverse directories where the number of directory entries is small enough to fit in one data block. You should be able to use `ls`, `stat`, and `mkdir` at this point.

Step 5: Add the ability to create files where the data is stored in a single data block. Implement `a1fs_truncate()` first, then `a1fs_write()` and `a1fs_read()`.

Step 6: Now expand your implementations so that files and directories fit in one extent of 2 or more blocks.

Step 7: Add the ability to remove files and directories.

Step 8: Expand your implementation to support multiple extents.

**Tip**: 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.

**Tip:** 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.

## Testing and debugging recommendations

You can use standard Unix tools (that you have learned in CSC209) to manipulate directories and files in a mounted a1fs image in order to test your implementation. System call tracing with `strace` can help understand what syscalls they invoke to access the file system. Use can, in general, use the behaviour of the host file system (ext4) as a reference - your a1fs should have the same observable behaviour for operations that a1fs needs to support. You can also write

your own C programs that invoke relevant syscalls directly. As a part of your submission, you will write a shell script that exercises the functionality of your file system.

You will find it useful to run `a1fs` under `gdb`:

```
gdb --args ./a1fs <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 to help understand what callbacks are invoked when you execute a file system operation (e.g. `mkdir`), in what order, and with what arguments. The debugger is also helpful in investigating crashes (e.g. segfaults). Please refer to `gdb` documentation for instructions on how to use it.

You might also find it useful to view the binary contents of your a1fs 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 a1fs 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 1 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 `a1fs` 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.

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()`.

## Limits and details

- The maximum number of inodes in the system is a parameter to `mkfs.a1fs`, the image size is also known to it, and the block size is `A1FS_BLOCK_SIZE` (4096 bytes). Many parameters of your file system can be computed from these three values.
- Your file system does not have to support more then 512 extents in a file.
- 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 what's necessary to store the inodes and the information about free and available blocks and inodes) in your `mkfs.a1fs` implementation.
- The maximum path component length is `A1FS_NAME_MAX` (252 bytes including the null terminator). This value is chosen to fit the directory entry structure into 256 bytes (see `a1fs.h`). Names stored in directory entries are null-terminated string so that you can use standard C string functions on them.
- The maximum full path length is `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.
- There are no limits on the size of files (except as dictated by the number of extents per file).
- There are no limits on the number of directory entries (except as dictated by the number of extents per file).
- There are no limits on the number of blocks in your file system (except for what can be addressed with 32-bit block pointers and 4096-byte block size).
- 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 128 inodes
- 10MiB size and 4096 inodes
- 2GiB size and 4096 inodes

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 complete 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 (`mkfs`) must also check that the image file is large enough to accommodate the requested number of inodes.

Other implementation notes:

- There is no need to physically store the "." and ".." directory entries - they can be manually listed by your `a1fs_readdir()` callback.
- 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 should only be allocated on demand.
- Read and write I/O should NOT be performed byte-by-byte (which is very inefficient); use `memcpy()`.
- Your implementation shouldn't use any floating point arithmetic. It is very easy to implement things like floor and ceiling division using integer arithmetic.

## Documentation

It is recommended that you include a README.txt file that describes any aspects of your code that does not work well. Code that works well and implements a subset of the functionality will get a higher mark than code that attempts to implement more functionality but doesn't work.

You will also submit a shell script (bash or sh) called `runit.sh` that demonstrates some functionality of your system. It should:

- create an image
- format the image
- mount the image
- perform a few operations on the file system such as creating a directory, displaying the contents of a directory, creating a file, adding data to a file (Keep it simple)
- unmount the image
- mount the image again and display some contents of the file system to show that the relevant state was saved to the disk image.

The purpose of this shell script is to show basic functionality. It is not meant to test corner cases or large files or directories.

## What to submit

Add all the starter files (obtained from MarkUs) to your a1b repository. Also add to your repository all additional source code files that you create as part of your implementation.

Your a1b repository will contain all the files necessary to compile and run `mkfs.a1fs` and `a1fs`. It will also include the `runit.sh` script. It may include a README file as described above. Do NOT add and commit virtual machine images, executables, .o or .d files, disk image files, or

any other unnecessary files - you will lose code style marks you do submit those. You are welcome to commit test code and other text files. You should use a `.gitignore` file to help ensure you only commit and push files you should.

## Useful links

libfuse GitHub repository: **https://github.com/libfuse/libfuse (https://github.com/libfuse/libfuse)**

FUSE API header file for the version we're using: **https://github.com/libfuse/libfuse/blob/fuse_2_9_bugfix/include/fuse.h (https://github.com/libfuse/libfuse/blob/fuse_2_9_bugfix/include/fuse.h)**

FUSE wiki: **https://github.com/libfuse/libfuse/wiki (https://github.com/libfuse/libfuse/wiki)**