CS3104: Practical 2 (File System) - Report

Matric Number: 170002815

4th November 2019

1 Overview

The objective of this practical was to implement a file system, which includes the support for files and directories structured as a hierarchical tree, as well as multiple common file system operations. The practical specification listed the use of FUSE and UnQLite as requirements, as well as making the data persistent for when the file system is not active. My approach for these challenges is discussed in the *Design & Implementation* section of this report.

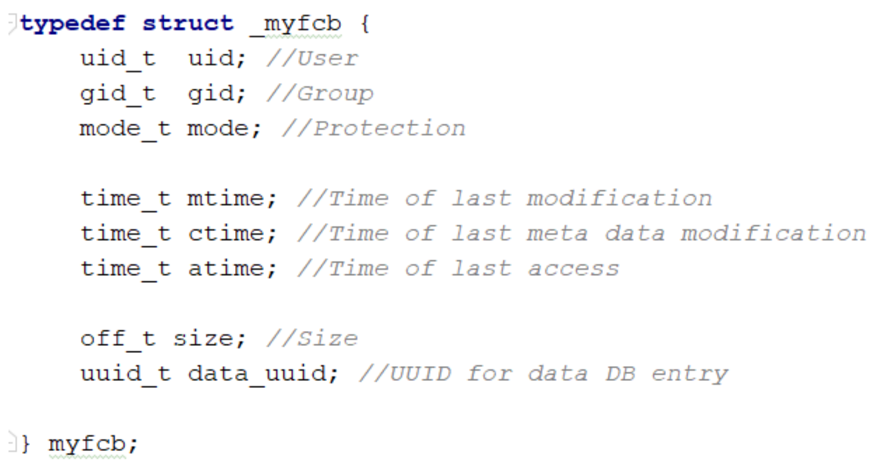
Testing of the program, and how completed extensions were approached are discussed in sections 3 and 4 respectively. The report is completed by an evaluation and conclusion, which reflect back on the learning objectives of the practical, and also evaluate how successful I have been in relation to the specification.

2 Design & Implementation

2.1 File Control Blocks (FCBs)

While completing research for this practical, I found the online resources for information on file control blocks to be quite limited, and therefore used multiple sources to determine my chosen structure.

Silberschatz's et al. Operating System Concepts [1] describes file control blocks to contain "information about the file, including ownership, permissions, and location of the file contents." - this definition agrees with the suggested FCB structure in the specification, leading me to use this as the foundation for my design, shown in *Figure 1* below.



```
typedef struct _myfcb {
    uid_t  uid; //User
    gid_t  gid; //Group
    mode_t mode; //Protection

    time_t mtime; //Time of last modification
    time_t ctime; //Time of last meta data modification
    time_t atime; //Time of last access

    off_t size; //Size
    uuid_t data_uuid; //UUID for data DB entry

} myfcb;
```

Figure 1: Chosen FCB Structure

It can be seen in *Figure 1* that the member `atime` is an addition to the struct in comparison to the one provided by the specification - this is to conform with Linux's `stat(2)`, which uses this attribute in its functionality.

2.2 Data Structures

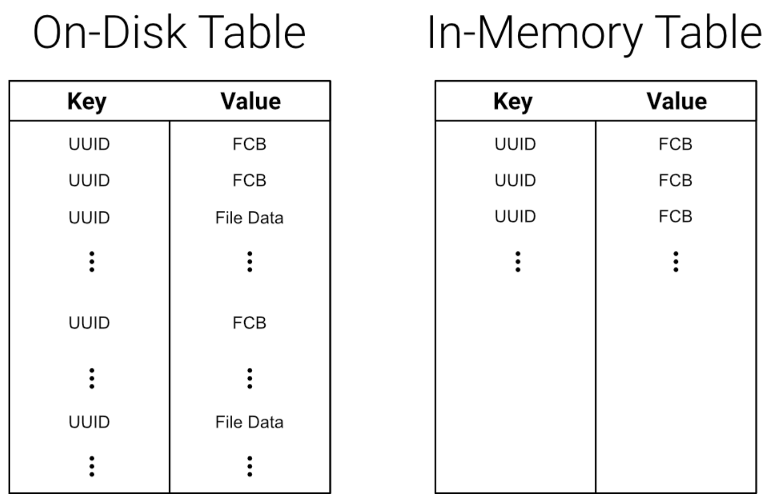
One of the most important decisions to make when completing this practical was which data structures to implement in order to create an efficient and effective file system, whilst also keeping the data persistent.

The specification listed the use of UnQLite for the on-disk storage of both FCBs and file data. By storing all of this information on-disk, all file system information and data can be accessed during its initialisation, even after system shutdown - this makes the data persistent and is discussed further in *Section 2.4*. The on-disk data structure is implemented as a database by UnQLite, and UUIDs are used as keys to match with the specification. A table representing the on-disk database can be seen in *Figure 2*.

For the in-memory data structures, I realised it was important to minimise overhead, but also maximise the total information included to reduce disk accesses, which are slower than memory accesses. Since caching was considered an extension for the practical, I decided to store every FCB in the program's memory - although this may be considered a huge overhead for my file system, it would avoid on-disk look-ups altogether for FCBs, therefore speeding up many common file system operations, such as `ls`, which has to read the FCB for every file in the specified path. By creating the in-memory structure, it would also have been easier to implement caching as an extension than if I had not cached any FCBs whatsoever.

After choosing to store all FCBs in memory, I had to decide on the optimal data structure to represent them - I considered an array, a linked list, a binary tree, and a hash table. An array was too limiting with its size having to be predetermined by the program before mounting the file system, and after serious consideration I decided to use a hash table, since its $\mathcal{O}(1)$ search complexity was far better for expansive file systems than a linked list's $\mathcal{O}(n)$ average search complexity or a binary tree's $\mathcal{O}(log(n))$ best case search complexity.

Since UnQLite was already being included in the program for the on-disk storage, I decided to use make use of the same database engine to implement my desired in-memory structure. The `UNQLITE_OPEN_IN_MEMORY` flag was used to initialise an in-memory database, as discussed in UnQLite's documentation [2] (N.B. `UNQLITE_OPEN_IN_MEMORY` is referenced as `UNQLITE_OPEN_MEM` on this page). I believe that UnQLite's in-memory database represents a hash table very well, as it provides a key-value store and also an $\mathcal{O}(1)$ search complexity [3]. Since the in-memory database was storing the same FCBs as the on-disk database, I decided to continue the consistency by using the same UUIDs in both databases as keys. The in-memory data structure can be seen alongside the on-disk data structure in *Figure 2*.



On-Disk Table

Key	Value
UUID	FCB
UUID	FCB
UUID	File Data
⋮	⋮
UUID	FCB
⋮	⋮
UUID	File Data
⋮	⋮

In-Memory Table

Key	Value
UUID	FCB
UUID	FCB
UUID	FCB
⋮	⋮

Figure 2: Data Structures

2.3 Hierarchical Directory Structure

One of the key requirements for this practical was to implement a tree-structured directory hierarchy. *Figure 3* displays an image taken from Silberschatz's et al. textbook [1], which shows an example of a tree-structured directory hierarchy, where the circular objects represent files, and the rectangular objects represent names.

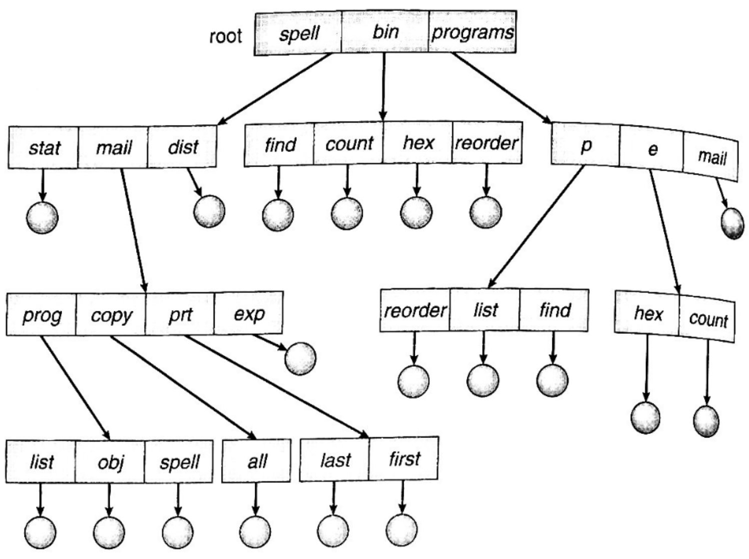


Figure 3: Tree-structured directory structure; taken from Silberschatz's et al. Operating System Concepts [1]

It can be seen in *Figure 3* that each directory needs to be able to other contain files or directories, which includes storing their names, and in the case of this practical, their file control blocks. Once again taking inspiration from Linux's file system, I decided to overcome this challenge by storing a list of dirents in each directory's data accessed using the key stored in its FCB.

A dirent, short for "directory entry", is a structure which represents information about a directory entry - a file or sub directory in the case of this practical. Ubuntu's man page on `readdir` [4] states that a Linux dirent contains the fields `d_ino`, `d_off`, `d_reclen` and `d_name`. To keep my file system implementation simple (due to time constraints), I decided to use only the fields `d_ino` and `d_name` in my dirent structure, which represent the inode number (FCB UUID in this case) and filename for each directory entry - these are named `fcb_uuid` and `name` respectively. I decided to set the maximum file name length to 256 bytes (including the string termination character) to keep consistency with linux. The definition of my dirent struct can be found in `myfs.h` at about line 40.

Given a directory's FCB, the process of accessing the FCB for any file or directory contained within the original directory involves the following steps:

1. Retrieve the directory's data stored using its FCB's `data_uuid` member as a key in the on-disk database. If there is no data, then there are no files or directories contained within this directory, and the search must be for a nonexistent file or directory.
2. Iterate through the retrieved data, reading each dirent's `name` member to check for a match. The `size` member of the original FCB can be used to avoid any segmentation faults caused by searches for nonexistent files or directories.
3. Once a match is found, the FCB can be retrieved using the dirent's `fcb_uuid` member as a key for the in-memory database.

Since dirents can represent both files and directories, storing a list of them in a directory's data creates the functionality of a tree-structured directory hierarchy for my implementation. Unfortunately the average search time for a sub directory or file contained within a directory is linear, however I decided to settle for this implementation, as most directories will not contain enough entries to seriously affect performance.

2.4 File System Initialisation & Persistent Data Retrieval

One of the largest challenges during this practical was the initialisation of the file system. The requirement of persistent data along with perhaps an excessive caching implementation meant that all previous files and directories stored needed to be retrieved during startup.

Following the directions of the specification, I used a well defined key (UUID) for the root FCB, which is defined using the preprocessor in `myfs.h`. By doing so, the root FCB can be fetched from the on-disk UnQLite database, and all other directories and files can be accessed using the procedure described in *Section 2.3*.

Because I decided to cache all file control blocks in memory, each one needed to be fetched from the persistent database during initialisation. To complete this task I wrote a recursive function named `add_children_to_mem(...)`, which is given an FCB and accesses all of its directory entries, adding each one to memory and recursively calling the function with each FCB as it is processed. The base cases for the method are when the directory has no entries, or when the provided FCB is for a file instead of a directory - these conditions can be checked by reading the `size` and `mode` members of the provided FCB. The code for this function can be seen at approx. lines 1850 to 1900 in `myfs.c`.

If the file system is being mounted for the first time, or in some other rare cases, there will be no stored root FCB in the on-disk database. In this case, `add_children_to_mem(...)` is not called, and a new root FCB is initialised instead.

2.5 File System Shutdown

The file system shutdown is a simple procedure, and only requires the closure of both the on-disk and in-memory databases, as well as the memory release of the cached root FCB, since it is stored outside of the in-memory database. According to UnQLite's documentation [2], all entries in an in-memory database are safely freed on its closure, meaning there was very little code for me to write to complete my file system's shutdown.

2.6 Global Pointers

In effort to reduce the number of global pointers used, as they are usually regarded negatively, I decided to only store pointers to the root FCB, the in-memory database, and the on-disk database. The root FCB is required for the file tree traversal, as the UUIDs used to retrieve any other FCB in the file tree can only be accessed if the roots FCB's stored dirents can be accessed.

Global pointers to the on-disk and in-memory databases are also kept, as this is the cleanest way to store this data while implementing UnQLite correctly. A UUID storing the zero UUID is also kept in the form of a global pointer, however this was used for debugging purposes only.

2.7 Auxiliary Functions

During the implementation of my program, I came across multiple tasks that needed to be repeated frequently. To reduce the volume of repeated code, and also aid in debugging, I decided to wrap these tasks in functions that could be used multiple times in multiple different places in my code. These functions mainly included interaction with the databases, however it also proved useful to write a function for checking an FCB's permissions and finding an FCB from a given path.

2.7.1 Database Interaction Functions

Eight functions used to interact with both the in-memory and on-disk databases are defined immediately after the global variables in `myfs.c` (approx. lines 20 to 120). There are three functions defined to fetch, store and delete an FCB using a given UUID, and there are three mirror functions that complete the same processes for any file/directory data. Two extra functions are defined, which fetch an FCB specifically from disk, and store an FCB specifically in memory - both of these are used exclusively during the file system's initialisation.

The file/directory data is only stored in the on-disk database, and therefore the auxiliary functions surrounding this data only need to access this database, making them fairly simple. A size is required for the fetch and store method, as the magnitude of the data is variable, and the return code of the database operation is returned for all methods. The advantage of wrapping these database operations in a function is two fold: if the storage method is needed to be changed, there is only a small amount of code that needs to be manipulated, and any extension involving alternative storage techniques (e.g. indexing) would have been easier to implement.

The FCB fetch, store and delete functions are slightly more complicated than those involving file/directory data, since there are potentially two databases to access rather than just one. The fetch method only needs to access the in-memory database, since all FCBs should be stored there, however the write and delete functions require accessing the on-disk database as well as the in-memory database. I decided to handle the in-memory database before the on-disk database for these operations, since it would facilitate a delayed write back extension, and also prevents FCBs existing in memory without being backed up on disk. This situation could happen if an error occurred with the database accessed second - in my implementation, a simple restart of the file system would fix any inconsistencies, however implementing with on-disk being accessed first could lead to data being lost.

2.7.2 FCB Permission Checking

The auxiliary function named `have_permission(...)` takes the input of an FCB and a individual character flag, and returns 1 if permissions can be granted to the requesting process, or 0 if otherwise, acting as a Boolean to "answer" the "question" posed by the name of the function.

The character flag is either 'r', 'w' or 'x', which is consistent with the common shorthand for "read", "write" and "execute", and decides which permission is to be checked on the provided FCB; although I could have implemented this function to accept multiple permissions, I decided to only allow single permission checking per function call, because most callers only require an individual permission (e.g. `read` only requires read permission), and it simplifies the function greatly.

The procedure of checking permissions in `have_permission(...)` was heavily influenced by independent research I completed during the practical. It was unknown to me before reading about permissions that there was an order of precedence in Linux, with the owner's permissions taking higher precedence than the group's permissions, which in turn takes higher precedence than others' permissions. That is to say that if the owner is the current user, and does not have a given permission, it does not matter about the group's and others' permissions, since the owner's permissions take highest precedence. This is implemented in my function by checking owner permissions first, followed by group permissions, and finally by others' permissions. I also included a check for the super user in `have_permission(...)`, as this overrules any other permission, however this was futile, as discussed in *Section 2.11.2*.

2.7.3 Fetching an FCB From a Path

A common task used in every file system operation is to retrieve a file control block from a given path; therefore I decided to define an auxiliary function named `get_fcb_from_path(...)` to complete this task. Unfortunately, completing some of the steps used to fetch an FCB inside a different function than the one completing a file system operation caused me some problems - fetching the UUID to point to the FCB and returning error codes (e.g. `ENOENT`). After some consideration, I settled for the function taking a path as a string and two user created buffers - one for the found FCB, and one for the UUID used to fetch the found FCB. 0 is returned, unless an error was found, in which case its code would be returned. This unfortunately involves callers to create two buffers and also check for an error code on function completion, however this is still far superior than copying out the function's contents multiple times in different locations in my code.

The process completed by `get_fcb_from_path(...)` is largely based off the procedure used to find a directory entry from its parent FCB, which is described in *Section 2.3*; however the procedure is repeated using a while loop, and the name to be found is tokenised from the path using `strtok(...)` with '/' as the delimiter. To keep consistency with Linux's file system operation descriptions in the man pages [4], I checked read permission on each directory in the path and checked for a file in the place of a directory in the path, returning the correct error codes in each circumstance.

2.8 File System Operation Handlers

The implementation of my file system needed to make use of the FUSE API, as described by the practical. In order for this API to work, a set of file system operation handlers needed to be defined and passed to FUSE. In order to implement my operation handlers correctly, I took inspiration from FUSE's documentation [5], the Linux man pages [4], and Harrison Middleton University's Computer Science resources [6]. Each operation handler is briefly discussed in the next few pages, and the code for all of the functions can be seen in `myfs.c` between approx. lines 300 and 1400. N.B. My implementation of each operation handler is prefixed with "myfs_" in `myfs.c`.

2.8.1 getattr(...)

Used to read a file or directory's attributes. Following advice from HMU's documentation [6], unknown fields given 0 or a "reasonable value" (e.g. number of hard links set to 1 since multiple hard links are not supported for my basic implementation). The general procedure for `getattr(...)` is described below.

1. Zero given buffer.
2. Find FCB from given path.
3. Fill given buffer with values from FCB.

2.8.2 readdir(...)

Used to read all entries in a directory. The general procedure for `readdir(...)` is described below.

1. Add '.' and '..' to buffer.
2. Fetch FCB for directory to read from given path.
3. Check for read permissions on fetched FCB, and check that it is a directory's FCB.
4. Fetch the directory data.
5. Loop through the data, adding each dirent to the buffer.

2.9 read(...)

Used to read the contents of a file. The general procedure for `read(...)` is described below.

1. Fetch FCB for file to read from given path.
2. Check for read permissions on fetched FCB, and check that it is a file's FCB.
3. Fetch the file data.
4. Copy the fetched data up to the requested size into the buffer. If the requested size of the read is greater than the size of the read available, copy what is available into the buffer.

2.9.1 create(...)

Used to create a file. Since FUSE will automatically call `get_attr(...)` on the path, there is no need to check that this `create(...)` is not duplicating a name inside of itself. The general procedure for `create(...)` is described below.

1. Divide given path into desired filename and path to its desired location.
2. Check that the filename does not exceed system limits.
3. Fetch FCB of desired parent directory using path formed in step 1.
4. Fetch data using UUID from parent directory's FCB.
5. Create new dirent and insert into parent directory data at end.
6. Store modified parent data.
7. Create and store FCB for new directory entry. New FCB is set using provided mode and the regular file mode flag.

2.9.2 mkdir(...)

Used to create a directory. The way I have implemented my file system means that this operation handler is almost identical to `create(...)`.

2.9.3 unlink(...)

Used to delete a file. Since FUSE will automatically call `get_attr(...)` on the path, there is no need to check the file to be deleted already exists. The general procedure for `unlink(...)` is described below. Although step 8 reorders the list of dirents, it does not matter since entries are printed alphabetically after calling `filler(...)` in `readdir(...)`.

1. Divide given path into name of file to remove and path to its parent directory.
2. Fetch parent directory FCB from path formed in step 1.
3. Check for write permissions on the fetched FCB.
4. Fetch FCB of file to remove & check it isn't for a directory.
5. Fetch parent FCB stored data using UUID from the FCB.
6. Find dirent matching file to remove in data fetched in step 5.
7. Delete the FCB entry for the file to delete from both databases using the key contained in the dirent found in step 6.
8. Delete the file data using the UUID stored in the FCB fetched in step 4.
9. Remove the dirent from the parent data. If the removed dirent is not the last dirent in the list, move the last dirent in the list into the empty space created by removing the dirent.
10. Update the parent's FCB to account for the new size of its data.
11. Store the new parent's FCB and data in the database(s).

2.9.4 rmdir(...)

Used to delete a directory. The way I have implemented my file system means that this operation handler is almost identical to `unlink(...)`.

2.9.5 utime(...)

Used to update times stored in a file or directory's metadata. The general procedure for `utime(...)` is described below.

1. Fetch FCB from given path.
2. Set data modification time (mtime) to that stated in the provided buffer, or the current system time if NULL (as specified in utime's man page [4]).
3. Set access time (atime) to that stated in the provided buffer, or the current system time if NULL (as specified in utime's man page [4]).
4. Set metadata modification time (ctime) to the current system time.
5. Write FCB back to databases.

2.9.6 write(...)

Used to write to a file. The general procedure for `write(...)` is described below.

1. Fetch FCB from given path.
2. Check write permission is given for FCB found in step 1.
3. Calculated requested new file size from offset and write size, checking that it is not greater than the system's limits.
4. Fetch required current file data into a buffer (i.e. up to offset).
5. Write new data into the buffer described in step 4, starting at the offset.
6. Update the times in the FCB fetched in step 1.
7. Store the updated FCB and file data back to database(s).

2.9.7 truncate(...)

Used to change a file's size. The general procedure for `truncate(...)` is described below.

1. Check new new size is not greater than the system's limits. The man page for `truncate(2)` suggests both `EINVAL` and `EFBIG` should be returned if this check shows a problem; since only one error code could be returned, I decided to use `EFBIG` since it seemed a more intuitive error name for the problem.
2. Check new size is not less than 0.
3. Fetch FCB from given path.
4. Check for no work (same size requested as is current).
5. Fetch the data associated with the FCB fetched in step 3.
6. Create a buffer with size of the requested size, and copy the data fetched in the previous step into it. There are two possibilities:
    - The buffer is smaller than the original data, and is a copy up until the buffer ends.
    - The buffer is larger than the original data, and contains all of it with 0s filling the extra space at the end of the buffer.
7. Store the new buffer using the key stored in the FCB fetched in step 3.
8. Update the size value in the FCB fetched in step 3, and store it.

2.9.8 chmod(...)

Used to change a file or directory's permissions. The general procedure for `chmod(...)` is described below.

1. Fetch the FCB and UUID from the given path.
2. Check the current user has permission. As described by `chmod(...)`'s man page, the user must either be the root user (user ID of 0), or they must be the owner of the file.
3. Edit the FCB's mode member to incorporate the new mode provided.
4. Write the FCB back to the database(s).

2.9.9 chown(...)

Used to change a file or directory's ownership. The man page for `chown(...)` suggests the caller must be the root user, unless they are the owner of the file and are changing the group to a group they are in. Unfortunately root user is not supported by FUSE, since all operations are run under the user who mounted the file system, and checking the groups for a user would require a password request, which is also not supported by FUSE. I have therefore implemented this function without any permission checking. The general procedure for `chown(...)` is described below.

1. Fetch FCB and UUID from given path.
2. Set new uid and gid in the FCB.
3. Write the FCB back to the database(s).

2.9.10 open(...)

Used to open a file. FUSE handles file descriptors, so this handler only needs to check permissions as described in `open(2)`'s man page [4]. The general procedure for `open(...)` is described below.

1. Fetch FCB from given path.
2. Compare flags provided by FUSE to the permissions in the fetched FCB, checking for permission errors.
3. Check the path doesn't lead to a directory and flags involve writing - this is to conform with Linux's man pages [4].

2.9.11 release(...)

Used to release any temporary data structures after file operations are complete. Since I handled all heap memory functions (e.g. `malloc(...)` and `free(...)`) in the other operation handlers, I needed to complete no actions for this handler.

2.9.12 flush(...)

Used to write any left-over data to disk. Since I have handled all writing in other operation handlers, I could leave this handler empty. I decided to keep the log writing function, as it kept consistency with other handlers.

2.10 Errors

During the initialisation of the file system and the execution of operations, errors may occur when working with the in-memory and on-disk databases. Errors may include the database currently being accessed by another call, permission errors, and the database running out of memory. These errors are described on the function pages of the UnQLite API [7].

Each error comes with a unique code, allowing for specific actions to be completed when each error occurs; an example of this in my code is during the file system initialisation, where the error code `UNQLITE_NOTFOUND` is returned, leading the to the creation of a new root FCB.

Database operation success also has a code (`UNQLITE_OK`), which I found to be very useful, since most errors required the same return code (`EIO`), making error checking very simple for operation handlers, as the OS has specific procedures to follow on the return of `EIO`. Errors during initialisation required a slightly more severe action, since a database not being opened correctly was fatal to the file system, and therefore I exited the program on a failure in these circumstances using the `exit(...)` function.

2.11 Limitations

Although my file system implementation is completely functional in relation to the specification, I did find some limitations during my design and implementation phase of this practical.

2.11.1 Documentation

The lack of documentation for FUSE was very apparent throughout this practical, and the author of HMU's FUSE documentation states this at the beginning of their writing [6].

The lack of documentation led to a lot of trial and error rather than considered design and proper implementation. An example of this is knowing which operation handlers are called for each Linux call, e.g. Linux's `rmdir` calls FUSE's `get_attr(...)`; I am still struggling to find confirmation whether I am completing the right procedure for each operation handler.

2.11.2 FUSE Caller IDs

Another limitation of FUSE is the inability to gather the operation caller's user ID and group ID from inside operation handlers. The commonly used `get_uid();` and `get_gid();` functions return the user ID and group ID of the user who mounted the file system, meaning that some operations' permission checking becomes redundant, e.g. checking for root user in `chown(...)`, as described in *Section 2.9.9*.

I have added the appropriate permission checking to each operation handler (except `chown(...)`), however my efforts seem futile as the user ID cannot be changed after file system mounting anyway.

2.11.3 Error Handling

As discussed in *Section 2.10*, my program handles errors during the initialisation process very aggressively by exiting the program when encountering even a small problem. I believe this is a fairly imposing limitation on my file system implementation, because some errors could be easily handled to avoid restarting the file system, and other errors could be avoided by rolling back the database(s), which would prevent any data being lost due to permanent failure to mount.

2.11.4 Memory Usage

Memory usage is probably the largest limitation of my basic implementation. As discussed in *Section 2.2*, every FCB is cached in memory, taking up an excessive amount of space, especially if the directory structure becomes very large. Alongside this problem, when reading data from the on-disk database, every single byte is read into memory, which is a limitation since memory is much more scarce than disk space, meaning my system's file size is limited by the amount of memory the machine it is running on has.

To overcome the limitation imposed by caching every FCB, I could adjust my implementation to cache just the root directory FCB and the FCBs of any entries in the root directory. Other FCBs could be cached as they are retrieved from the on-disk database, freeing them after a certain time has elapsed without using them, or when FUSE's `release(...)` is called.

To overcome the limitation imposed by reading the entirety of a file/directory's data into memory in one go, indexed or hybrid allocation could be implemented to only read the required blocks of arbitrary size into memory. This would also increase the file size limitation.

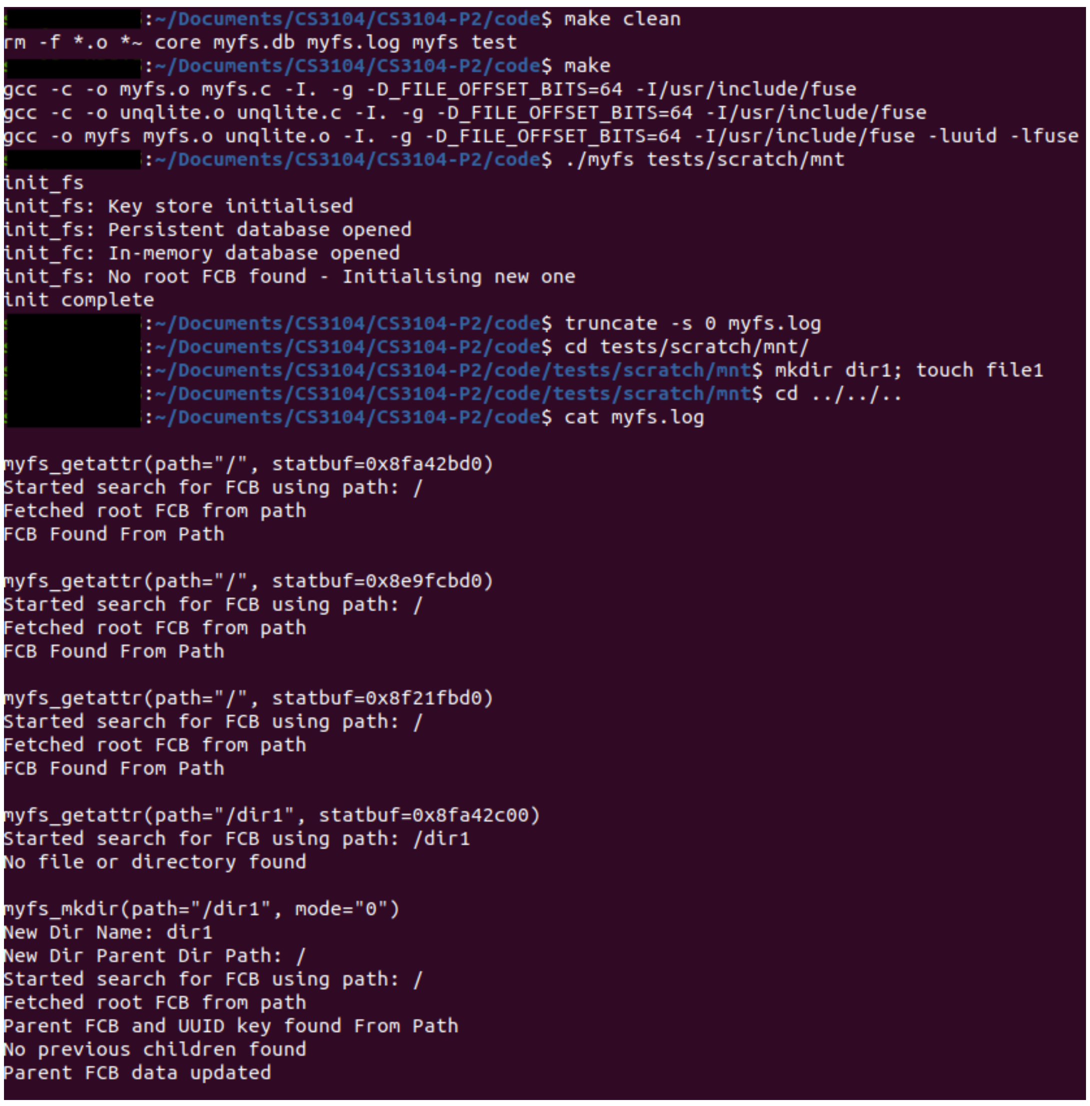
A small limitation of my program in relation to memory usage is my implementation of file/directory names. Every name, no matter the length, is stored in a 256 byte char array - this is a cause for fragmentation, as there is usually a lot of wasted space. A solution to this limitation would be to store the length of the name in my dirent structure, allowing me to implement a varible size. I decided this was a very complicated process for a fairly minute optimisation, and therefore decided to not implement this into my practical.

3 Testing

3.0.1 Log File

During the operation of my file system outside of debug mode, print statements will not be registered, and therefore a log file had to be used. The starter code provided the opening mechanism for the file, and also added some simple writes to describe which operation handlers are called when and which arguments they are supplied with.

I have expanded on the supplied starter code to make my program produce a trace of the operations completed - this can be seen in *Figure 4* below.



```
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make clean
rm -f *.o *~ core myfs.db myfs.log myfs test
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make
gcc -c -o myfs.o myfs.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -c -o unqlite.o unqlite.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -o myfs myfs.o unqlite.o -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -luuid -lfuse
[redacted]:~/Documents/CS3104/CS3104-P2/code$ ./myfs tests/scratch/mnt
init_fs
init_fs: Key store initialised
init_fs: Persistent database opened
init_fc: In-memory database opened
init_fs: No root FCB found - Initialising new one
init complete
[redacted]:~/Documents/CS3104/CS3104-P2/code$ truncate -s 0 myfs.log
:~/Documents/CS3104/CS3104-P2/code$ cd tests/scratch/mnt/
:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ mkdir dir1; touch file1
:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ cd ../../..
:~/Documents/CS3104/CS3104-P2/code$ cat myfs.log

myfs_getattr(path="/", statbuf=0x8fa42bd0)
Started search for FCB using path: /
Fetched root FCB from path
FCB Found From Path

myfs_getattr(path="/", statbuf=0x8e9fcbd0)
Started search for FCB using path: /
Fetched root FCB from path
FCB Found From Path

myfs_getattr(path="/", statbuf=0x8f21fbd0)
Started search for FCB using path: /
Fetched root FCB from path
FCB Found From Path

myfs_getattr(path="/dir1", statbuf=0x8fa42c00)
Started search for FCB using path: /dir1
No file or directory found

myfs_mkdir(path="/dir1", mode="0")
New Dir Name: dir1
New Dir Parent Dir Path: /
Started search for FCB using path: /
Fetched root FCB from path
Parent FCB and UUID key found From Path
No previous children found
Parent FCB data updated
```

Figure 4: Log File Displaying a Trace of the File System Operations

3.0.2 Debug Mode

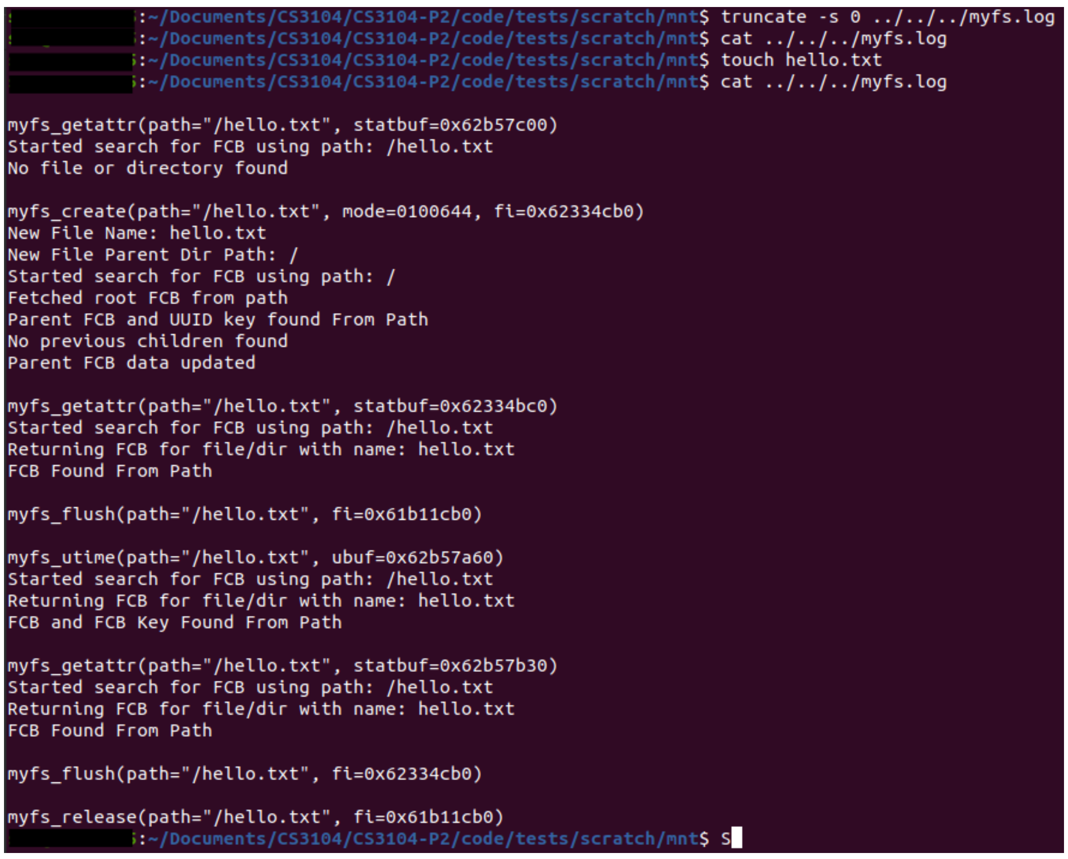
Although the log file was very useful for debugging most issues with my program, some problems were much easier to understand if the output of the C code could be seen - in these circumstances, I used the ”-d” flag when mounting my file system to start FUSE in debug mode.

An example of when debug mode proved useful for me was when I was mallocing the size of a size, providing me with a corrupted top size, however this error was only visible when running my program in debug mode, as running the program in standard mode only provided me with the ambiguous ”Transport endpoint not connected” error.

3.0.3 Terminal Commands

A simple, yet effective, way to test my file system implementation was to complete some simple tasks inside of the terminal. Bash commands such as `ls` and `mkdir` could be used to test specific operation handlers from `myfs.c`, and check their individual output to the log file - this is preferable to running several commands before checking the log, as it can be seen which operation handlers are completed and what actions they take.

I used too many commands during testing to include in this report, however an example of me testing my file system's `create(...)` operation handler and checking its log output is shown in *Figure 5* below.



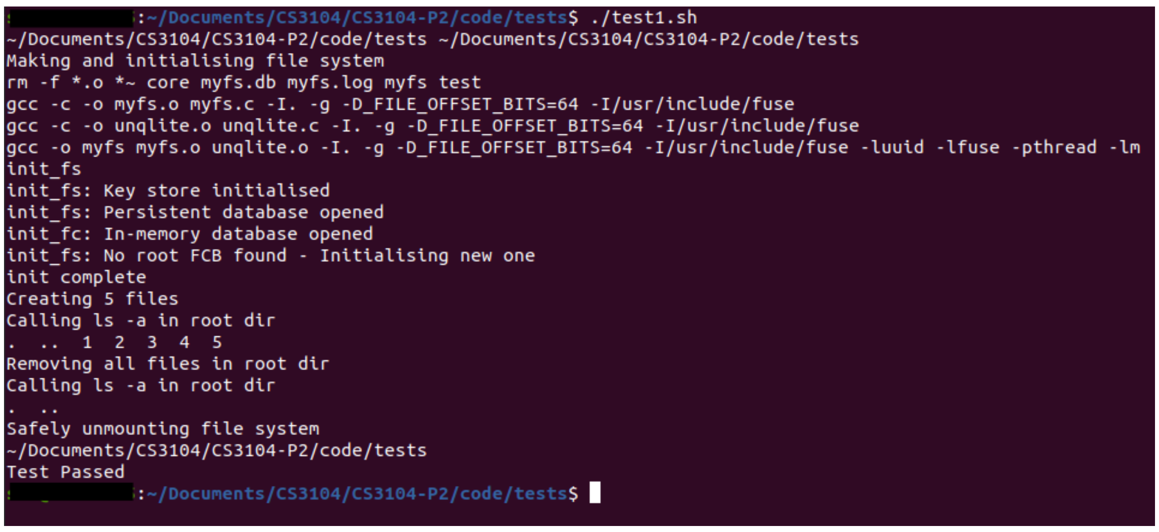
```
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ truncate -s 0 ../../../myfs.log
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ cat ../../../myfs.log
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ touch hello.txt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ cat ../../../myfs.log

myfs_getattr(path="/hello.txt", statbuf=0x62b57c00)
Started search for FCB using path: /hello.txt
No file or directory found

myfs_create(path="/hello.txt", mode=0100644, fi=0x62334cb0)
New File Name: hello.txt
New File Parent Dir Path: /
Started search for FCB using path: /
Fetched root FCB from path
Parent FCB and UUID key found From Path
No previous children found
Parent FCB data updated

myfs_getattr(path="/hello.txt", statbuf=0x62334bc0)
Started search for FCB using path: /hello.txt
Returning FCB for file/dir with name: hello.txt
FCB Found From Path

myfs_flush(path="/hello.txt", fi=0x61b11cb0)

myfs_utime(path="/hello.txt", ubuf=0x62b57a60)
Started search for FCB using path: /hello.txt
Returning FCB for file/dir with name: hello.txt
FCB and FCB Key Found From Path

myfs_getattr(path="/hello.txt", statbuf=0x62b57b30)
Started search for FCB using path: /hello.txt
Returning FCB for file/dir with name: hello.txt
FCB Found From Path

myfs_flush(path="/hello.txt", fi=0x62334cb0)

myfs_release(path="/hello.txt", fi=0x61b11cb0)
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$
```

Figure 5: Example of Testing Using Terminal Commands.

3.0.4 Shell Scripts

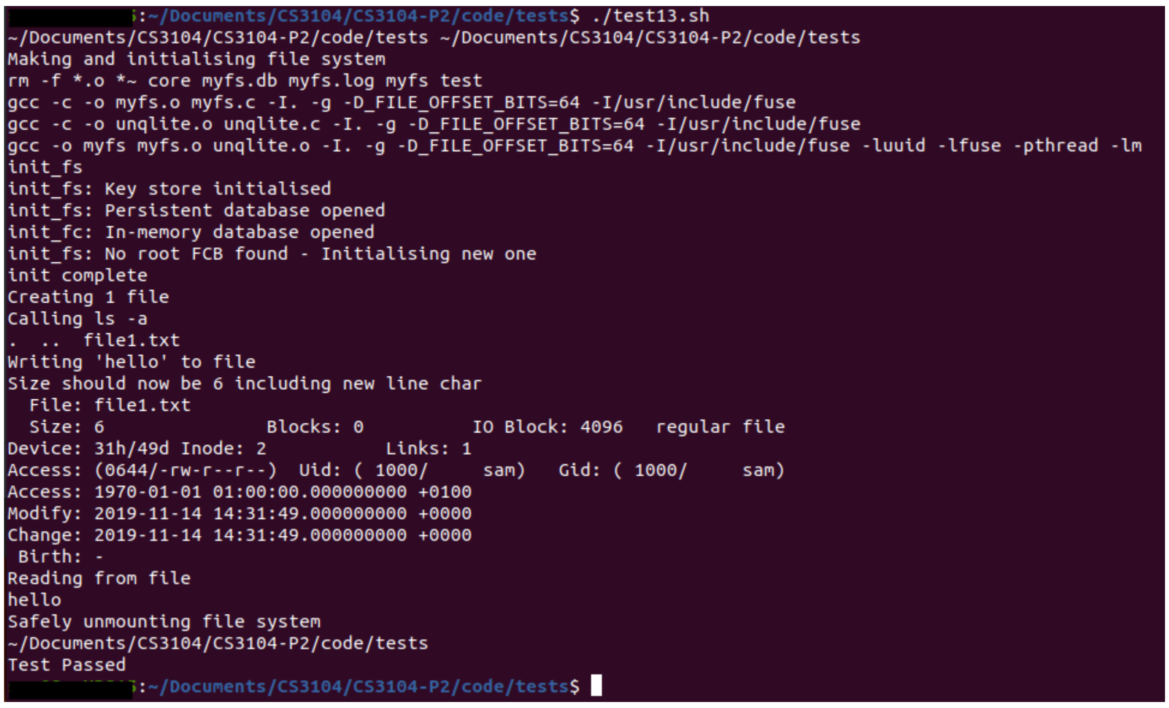
Although inputting individual terminal commands was an excellent way of testing specific operation handlers for my program, using this method to test more complex tasks, such as creating 5 files and then removing them, was very time consuming. To overcome this problem, I created shell scripts to complete several terminal commands for me, including the setup and shutdown of the file system. Unfortunately these shell scripts cannot be viewed, as the practical specification requires only `myfs.c`, `myfs.h` and this report to be uploaded, however a description of each test is provided on the next page, and a few example outputs can be seen in *Figure 6*, *Figure 7* and *Figure 8*.

- **Test 1 - Test 5**: Testing file creation and deletion functionality. Different tests remove different files to check that removal is being completed properly with no errors. Test 4 creates 50 files, testing the expansiveness of my file system. The output of Test 1 can be seen in *Figure 6*.
- **Test 6 - Test 9**: Similar tests as Tests 1 - 5, except with directories instead of files.
- **Test 10**: Testing the hierarchical directory structure of my file system by creating a file inside a directory, inside another directory, which is inside yet another directory in the root directory.
- **Test 11**: Testing the persistent functionality of my file system. This test creates two directories, each with a single file inside them. The file system is then unmounted and remounted, followed by a check for the directories and files still existing.
- **Test 12**: Testing that directories can hold both files and directories at the same time.
- **Test 13**: Tests functionality of reading/writing from/to a file, as well as gathering the meta-data correctly. The output of this test can be seen in *Figure 7*.
- **Test 14**: Tests rejection of creation of file/directory when name is already taken. The output of this test can be seen in *Figure 8*.
- **Test 15**: Tests rejection of creation of file/directory when name is too long.
- **Test 16**: Tests permission modification, as well as permission rejection for read and write.
- **Test 17**: Tests user modification.
- **Test 18**: Tests truncation to a smaller and larger size.
- **Test 19**: Tests rejection of `rmdir(...)` on a file, and `unlink(...)` on a directory.



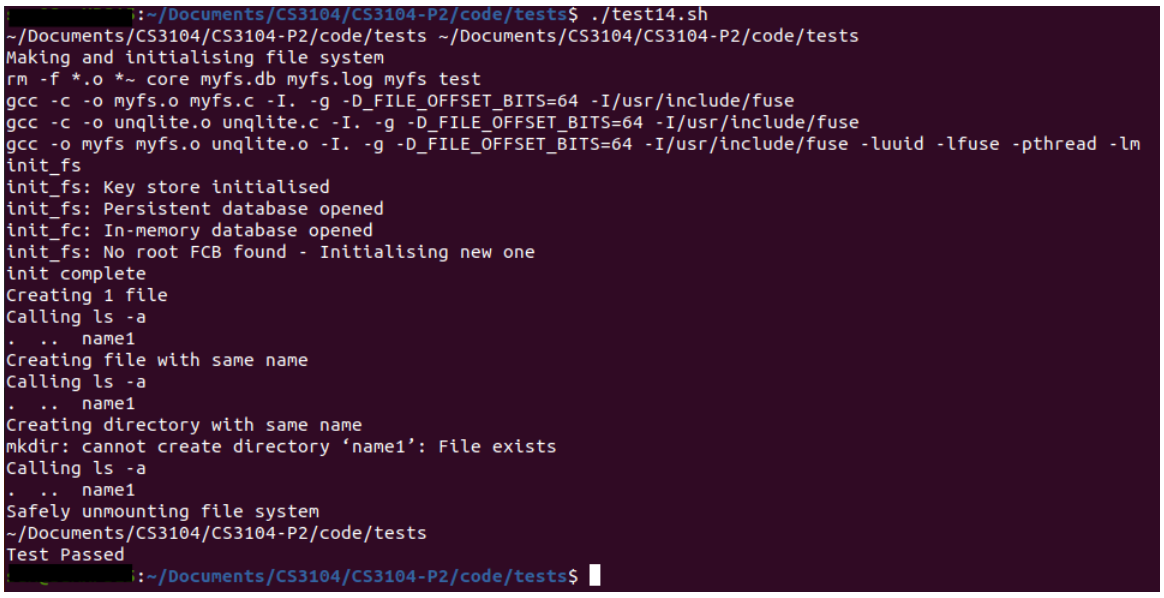
```
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests$ ./test1.sh
~/Documents/CS3104/CS3104-P2/code/tests ~/Documents/CS3104/CS3104-P2/code/tests
Making and initialising file system
rm -f *.o *~ core myfs.db myfs.log myfs test
gcc -c -o myfs.o myfs.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -c -o unqlite.o unqlite.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -o myfs myfs.o unqlite.o -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -luuid -lfuse -pthread -lm
init_fs
init_fs: Key store initialised
init_fs: Persistent database opened
init_fc: In-memory database opened
init_fs: No root FCB found - Initialising new one
init complete
Creating 5 files
Calling ls -a in root dir
.  ..  1  2  3  4  5
Removing all files in root dir
Calling ls -a in root dir
.  ..
Safely unmounting file system
~/Documents/CS3104/CS3104-P2/code/tests
Test Passed
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests$
```

Figure 6: Output of Test 1 (`test1.sh`)



```
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests$ ./test13.sh
~/Documents/CS3104/CS3104-P2/code/tests ~/Documents/CS3104/CS3104-P2/code/tests
Making and initialising file system
rm -f *.o *~ core myfs.db myfs.log myfs test
gcc -c -o myfs.o myfs.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -c -o unqlite.o unqlite.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -o myfs myfs.o unqlite.o -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -luuid -lfuse -pthread -lm
init_fs
init_fs: Key store initialised
init_fs: Persistent database opened
init_fc: In-memory database opened
init_fs: No root FCB found - Initialising new one
init complete
Creating 1 file
Calling ls -a
.  ..  file1.txt
Writing 'hello' to file
Size should now be 6 including new line char
  File: file1.txt
  Size: 6               Blocks: 0          IO Block: 4096   regular file
Device: 31h/49d Inode: 2           Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/     sam)   Gid: ( 1000/     sam)
Access: 1970-01-01 01:00:00.000000000 +0100
Modify: 2019-11-14 14:31:49.000000000 +0000
Change: 2019-11-14 14:31:49.000000000 +0000
 Birth: -
Reading from file
hello
Safely unmounting file system
~/Documents/CS3104/CS3104-P2/code/tests
Test Passed
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests$ 
```

Figure 7: Output of Test 13 (`test13.sh`)



```
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests$ ./test14.sh
~/Documents/CS3104/CS3104-P2/code/tests ~/Documents/CS3104/CS3104-P2/code/tests
Making and initialising file system
rm -f *.o *~ core myfs.db myfs.log myfs test
gcc -c -o myfs.o myfs.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -c -o unqlite.o unqlite.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -o myfs myfs.o unqlite.o -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -luuid -lfuse -pthread -lm
init_fs
init_fs: Key store initialised
init_fs: Persistent database opened
init_fc: In-memory database opened
init_fs: No root FCB found - Initialising new one
init complete
Creating 1 file
Calling ls -a
.  ..  name1
Creating file with same name
Calling ls -a
.  ..  name1
Creating directory with same name
mkdir: cannot create directory ‘name1’: File exists
Calling ls -a
.  ..  name1
Safely unmounting file system
~/Documents/CS3104/CS3104-P2/code/tests
Test Passed
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests$ 
```

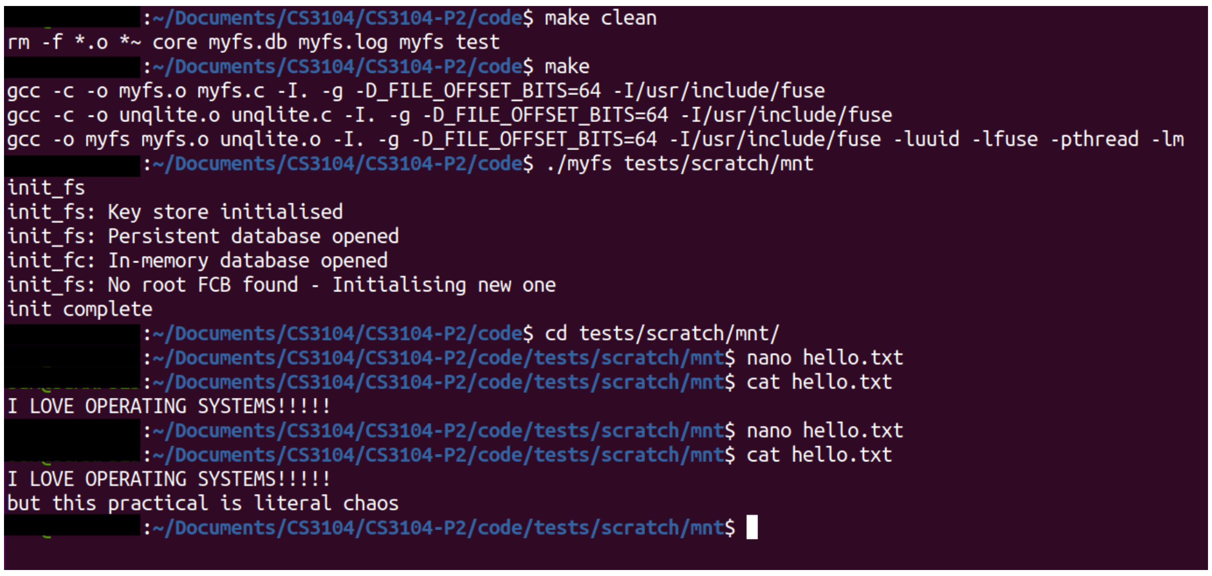
Figure 8: Output of Test 14 (`test14.sh`)

3.0.5 Test Programs

To further the testing of my file system, I decided to use the nano text editor and a C program to interact with the file system. Both of these testing methods could make use of the `open(...)` operation handler, which I was not able to test with simple Bash commands.

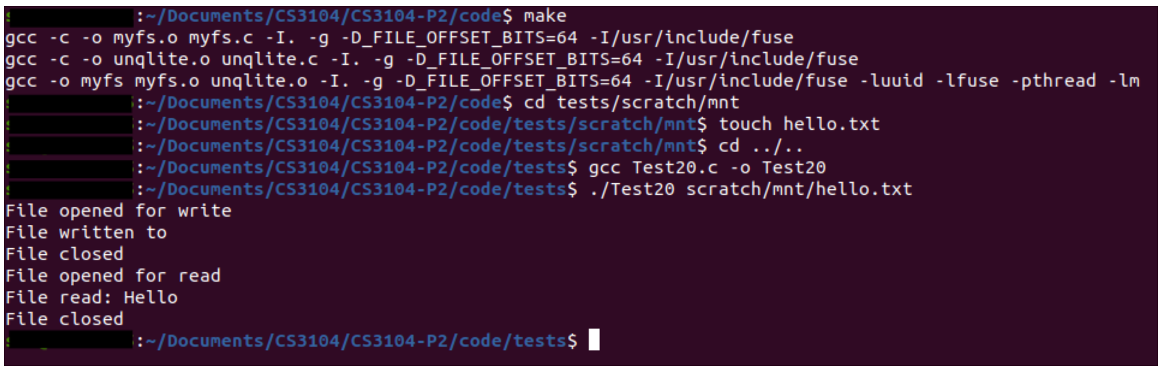
To test my file system implementation with the nano text editor, I decided to complete some simple read and writes - an example of this can be seen in *Figure 9*.

To test my file system implementation with a C program, I decided to write a program to open a file and then write to and read from it. The output of this program can be seen in *Figure 10*.



```
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make clean
rm -f *.o *~ core myfs.db myfs.log myfs test
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make
gcc -c -o myfs.o myfs.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -c -o unqlite.o unqlite.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -o myfs myfs.o unqlite.o -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -luuid -lfuse -pthread -lm
[redacted]:~/Documents/CS3104/CS3104-P2/code$ ./myfs tests/scratch/mnt
init_fs
init_fs: Key store initialised
init_fs: Persistent database opened
init_fc: In-memory database opened
init_fs: No root FCB found - Initialising new one
init complete
[redacted]:~/Documents/CS3104/CS3104-P2/code$ cd tests/scratch/mnt/
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ nano hello.txt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ cat hello.txt
I LOVE OPERATING SYSTEMS!!!!!
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ nano hello.txt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ cat hello.txt
I LOVE OPERATING SYSTEMS!!!!!
but this practical is literal chaos
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ 
```

Figure 9: Example Test Output Using nano Text Editor



```
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make
gcc -c -o myfs.o myfs.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -c -o unqlite.o unqlite.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -o myfs myfs.o unqlite.o -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -luuid -lfuse -pthread -lm
[redacted]:~/Documents/CS3104/CS3104-P2/code$ cd tests/scratch/mnt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ touch hello.txt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ cd ../..
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests$ gcc Test20.c -o Test20
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests$ ./Test20 scratch/mnt/hello.txt
File opened for write
File written to
File closed
File opened for read
File read: Hello
File closed
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests$ 
```

Figure 10: Example Test Output Using C Program

4 Extensions

There were many possible extensions for this practical, and the way I designed my implementation meant that multiple could be implemented without starting from scratch (e.g. caching), however because I wanted to make sure my basic implementation was to an excellent standard, I decided to implement two relatively small extensions - `open(...)` and `rename(...)`.

4.1 open(...)

There was some ambiguity around whether the `open(...)` operation handler was part of the basic specification for this practical, or whether it was considered an extension; however since I have implemented properly with permission checks to match the Linux man pages [4], I felt it appropriate to describe it in this section of the report.

Permission checks are completed in relation to the flags provided as an argument, and the specific scenario of opening a directory with the write flag specified is also checked for, which keeps consistency with Linux's man pages [4]. The procedure of `open(...)` is described in *Section 2.9.10* of this report, and *Figure 9* and *Figure 10* show that the operation handler performs properly when utilised by different programs.

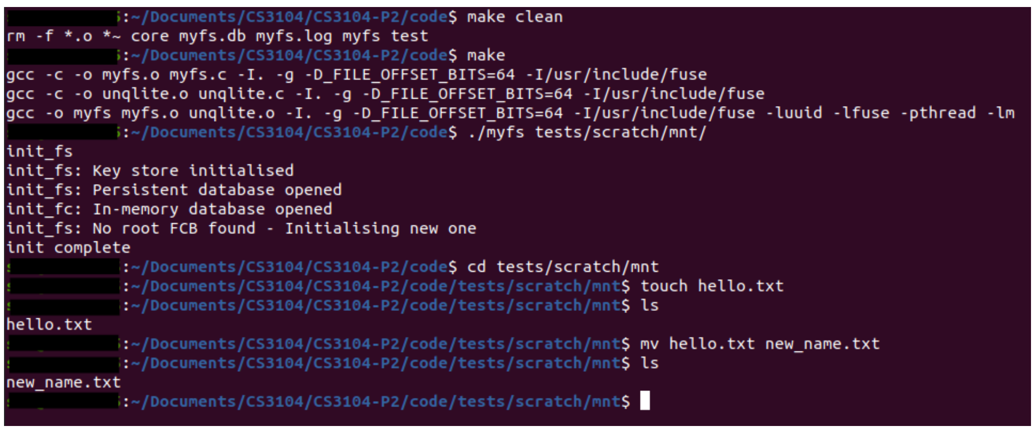
4.2 rename(...)

The file system operation handler `rename(...)` matches up to Linux's `rename(2)` operation, and is used to change the name and/or location of a file or directory. FUSE's definition of the handler takes two parameters - the old file path and the new file path. This file handler is the most complicated one I have implemented, making it a suitable extension to show off my skills I have learnt during this practical. The general procedure of `rename(...)` is described below, and images of the operation handler being used in different scenarios by the `mv` command can be seen in *Figure 11*, *Figure 12*, *Figure 13* and *Figure 14* which show a file being renamed, a file being moved between directories, a directory being moved, and permission being rejected respectively.

1. Copy the old file path and split it into the file's name and parent directory.
2. Repeat step 1 with the new file path.
3. Fetch the FCBs for both the old and new parent directories to check write permissions.
4. Fetch the old parent directory's data and find the dirent with the name matching that from step 1.

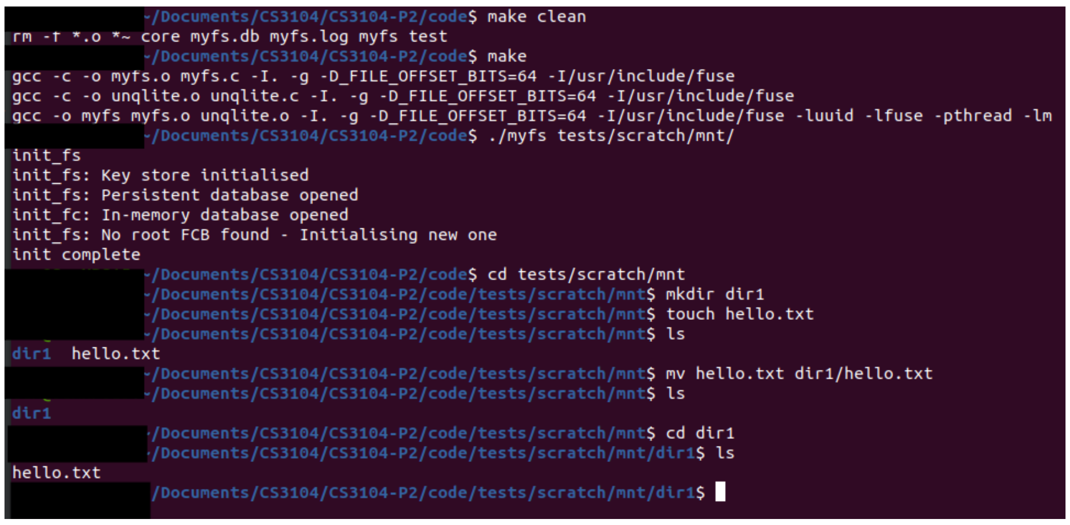
   If the new parent directory is the same as the old parent directory, a shortcut is completed here, which simply replaces the name in this dirent, then stores the data, completing the desired renaming and reducing the amount of completed computation.
5. Copy the UUID from the dirent found in step 4 and the name found in step 2 into a new dirent.
6. Remove the dirent found in step 4 from the old parent directory's data, therefore removing the file/directory from the old parent directory.
7. Update the old parent directory's FCB to represent its new size, and store this FCB, as well as the modified data.
8. Fetch the new parent directory's data and add the dirent created in step 5 to the end of it, therefore adding the file/directory with the new name to the new parent directory.
9. Update the new parent directory's FCB to represent its new size, and store this FCB, as well as the modified data.

Note: The lab clients will run `rename(...)` twice when a file is moved into a directory, providing an error to suggest there is no file that exists, however the operation is completed properly. I have tested my `rename(...)` operation handler on two other Linux machines with no errors, and could not find anything online about this problem.



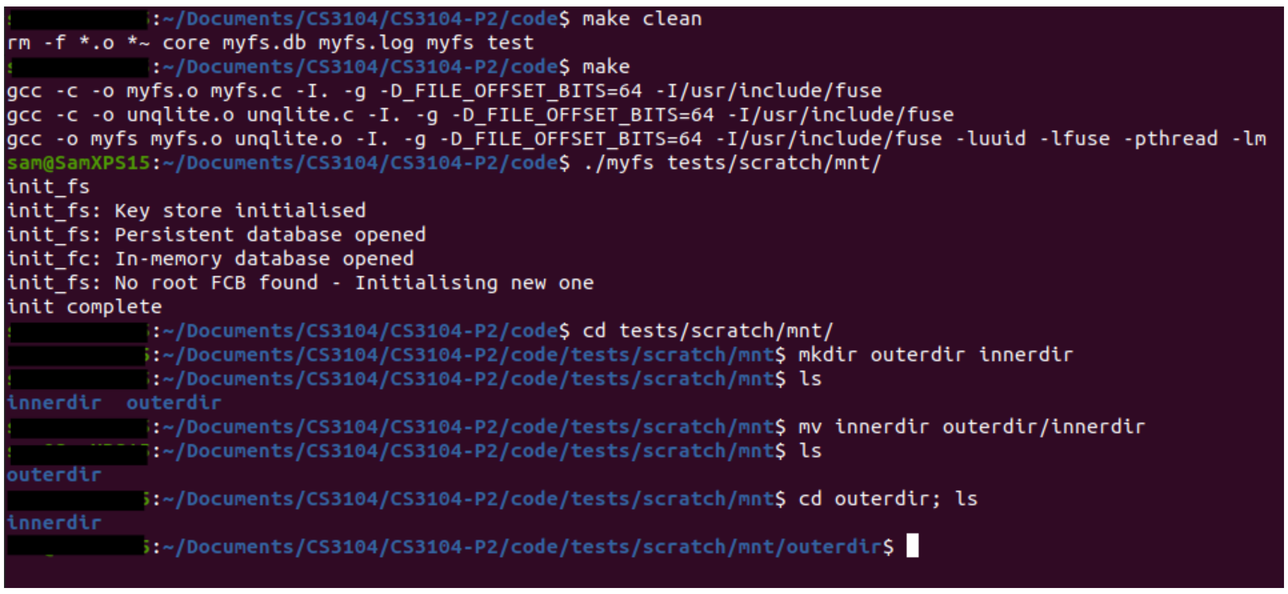
```
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make clean
rm -f *.o *~ core myfs.db myfs.log myfs test
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make
gcc -c -o myfs.o myfs.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -c -o unqlite.o unqlite.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -o myfs myfs.o unqlite.o -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -luuid -lfuse -pthread -lm
[redacted]:~/Documents/CS3104/CS3104-P2/code$ ./myfs tests/scratch/mnt/
init_fs
init_fs: Key store initialised
init_fs: Persistent database opened
init_fc: In-memory database opened
init_fs: No root FCB found - Initialising new one
init complete
[redacted]:~/Documents/CS3104/CS3104-P2/code$ cd tests/scratch/mnt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ touch hello.txt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ ls
hello.txt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ mv hello.txt new_name.txt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ ls
new_name.txt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ 
```

Figure 11: Example of a File Being Renamed Using `rename(...)`



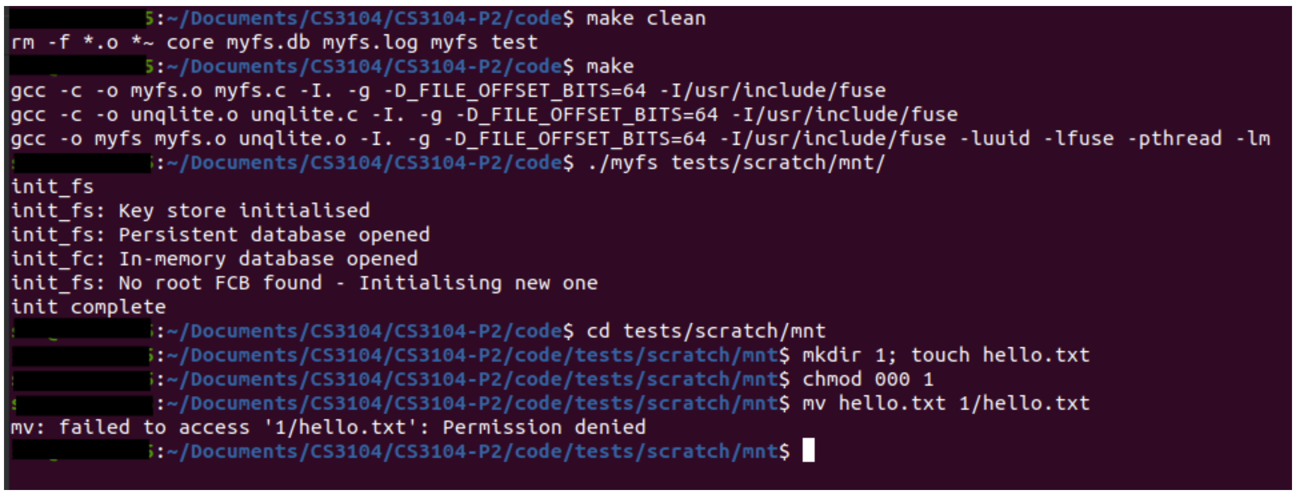
```
[redacted]~/Documents/CS3104/CS3104-P2/code$ make clean
rm -f *.o *~ core myfs.db myfs.log myfs test
[redacted]~/Documents/CS3104/CS3104-P2/code$ make
gcc -c -o myfs.o myfs.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -c -o unqlite.o unqlite.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -o myfs myfs.o unqlite.o -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -luuid -lfuse -pthread -lm
[redacted]~/Documents/CS3104/CS3104-P2/code$ ./myfs tests/scratch/mnt/
init_fs
init_fs: Key store initialised
init_fs: Persistent database opened
init_fc: In-memory database opened
init_fs: No root FCB found - Initialising new one
init complete
[redacted]~/Documents/CS3104/CS3104-P2/code$ cd tests/scratch/mnt
[redacted]~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ mkdir dir1
[redacted]~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ touch hello.txt
[redacted]~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ ls
dir1  hello.txt
[redacted]~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ mv hello.txt dir1/hello.txt
[redacted]~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ ls
dir1
[redacted]~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ cd dir1
[redacted]~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt/dir1$ ls
hello.txt
[redacted]/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt/dir1$ 
```

Figure 12: Example of a File Being Moved Between Directories Using `rename(...)`



```
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make clean
rm -f *.o *~ core myfs.db myfs.log myfs test
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make
gcc -c -o myfs.o myfs.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -c -o unqlite.o unqlite.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -o myfs myfs.o unqlite.o -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -luuid -lfuse -pthread -lm
sam@SamXPS15:~/Documents/CS3104/CS3104-P2/code$ ./myfs tests/scratch/mnt/
init_fs
init_fs: Key store initialised
init_fs: Persistent database opened
init_fc: In-memory database opened
init_fs: No root FCB found - Initialising new one
init complete
[redacted]:~/Documents/CS3104/CS3104-P2/code$ cd tests/scratch/mnt/
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ mkdir outerdir innerdir
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ ls
innerdir  outerdir
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ mv innerdir outerdir/innerdir
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ ls
outerdir
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ cd outerdir; ls
innerdir
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt/outerdir$ 
```

Figure 13: Example of a Directory Being Moved Between Directories Using `rename(...)`



```
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make clean
rm -f *.o *~ core myfs.db myfs.log myfs test
[redacted]:~/Documents/CS3104/CS3104-P2/code$ make
gcc -c -o myfs.o myfs.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -c -o unqlite.o unqlite.c -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse
gcc -o myfs myfs.o unqlite.o -I. -g -D_FILE_OFFSET_BITS=64 -I/usr/include/fuse -luuid -lfuse -pthread -lm
[redacted]:~/Documents/CS3104/CS3104-P2/code$ ./myfs tests/scratch/mnt/
init_fs
init_fs: Key store initialised
init_fs: Persistent database opened
init_fc: In-memory database opened
init_fs: No root FCB found - Initialising new one
init complete
[redacted]:~/Documents/CS3104/CS3104-P2/code$ cd tests/scratch/mnt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ mkdir 1; touch hello.txt
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ chmod 000 1
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ mv hello.txt 1/hello.txt
mv: failed to access '1/hello.txt': Permission denied
[redacted]:~/Documents/CS3104/CS3104-P2/code/tests/scratch/mnt$ 
```

Figure 14: Example Permission Being Rejected When Using `rename(...)`

5 Evaluation & Conclusion

Overall I believe I have been successful with this practical - I have implemented the basic specification to a high standard, and have completed both testing and extensions to accompany my work.

Reflecting on the learning objectives for this practical listed in the specification, I have expanded my knowledge of how to implement a file system and the concepts involved in that process. I have also improved my C programming skills.

Given more time, I would enjoy implementing a few more extensions, including a proper caching system and indexed/hybrid allocation, as discussed in the *Limitations* section of this report.

References

[1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. John Wiley Sons, 1 2005.

[2] Mrad Chems Eddine and Symisc Systems. Unqlite c/c++ api reference - database engine handle.

[3] Mrad Chems Eddine and Symisc Systems. Unqlite - distinctive features.

[4] *Linux User's Manual*.

[5] libfuse api documentation.

[6] Hmu: Fuse documentation.

[7] Mrad Chems Eddine and Symisc Systems. unqlite c/c api reference - list of functions.