CS3104 – Operating Systems

Assignment: P2 – A filesystem in userspace

Deadline: 29 November 2017 *Credits*: 60% of coursework mark

MMS is the definitive source for deadline and credit details

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

Aim / Learning objectives

The purpose of this assignment is three-fold:

- to give you experience of implementing a simple file system;
- to increase your experience of system level programming in C;
- to reinforce your understanding of file system implementation concepts.

Requirements

In this coursework exercise you will implement a simple POSIX file system. You are **required to use the C programming language** to write a FUSE user-level file system that utilises the 'unqlite' key-value store (http://unqlite.org/) for persistent storage.

The features required of your implementation are as follows:

- support for a tree-structured directory hierarchy
- directories that may contain both files and sub-directories
- file and directory meta-data including size, owner, group, mode and data/meta-data modification times
- arbitrary length file names (up to the system limits)
- file and directory creation, update and deletion
- variable length files and directories (although you may impose modest limits on both to simplify implementation)
- file truncation
- file and directory meta-data (including modification time, uid, gid, size and file permissions)
- logging of individual file system operations to file (this is already implemented in the starter code).

Source code and getting started

You are provided with starter code implementing the 'myfs' file system. This file system supports a single directory containing up to one file with up to 1000 bytes of data. The starter code can be found here:

http://studres.cs.st-andrews.ac.uk/CS3104/Practicals/P2-Filesystem/code.tgz

From a Linux lab client or your host server, unpack, compile and run the code as follows:

```
$ tar xzvf /cs/studres/CS3104/Practicals/P2-Filesystem/code.tgz
$ cd code/
$ make
$ mkdir /cs/scratch/<username>/mnt
$ ./myfs /cs/scratch/<username>/mnt
```

The myfs user-level file system is now mounted at '/cs/scratch/<username>/mnt'. Execute some file system operations:

```
$ ls –la /cs/scratch/<username>/mnt
$ stat /cs/scratch/<username>/mnt
$ echo "hello!" > /cs/scratch/<username>/mnt/afile.txt
$ cat /cs/scratch/<username>/mnt/afile.txt
$ ls –la /cs/scratch/<username>/mnt/afile.txt
$ stat /cs/scratch/<username>/mnt/afile.txt
```

Unmount the file systems as follows:

```
$ fusermount -u /cs/scratch/<username>/mnt
```

The fuse operations are logged in 'myfs.log'. Note that when the file system is mounted any existing log file will be overwritten. The starter code consists of a number of source files, including:

- myfs.c and myfs.h – the main function, definitions, initialisation code and fuse operation implementations. You should modify these two files.
- unqlite.c – the key-value store implementation.

Hints

During development, it is easy to corrupt the file system and fail to mount it again. You can easily restart by deleting the unqlite database file created by your program. If you get the message "Transport endpoint not connected", it means that your filesystem crashed and will need to be restarted. It can be useful to script some common operations to test your implementation during development.

Submission

You are required to submit an archive file containing ***three*** files. The first two will contain your source code in 'myfs.c' and 'myfs.h' while the third will contain your report in ***PDF***. The report must explain the design and implementation of your filesystem and your approach to testing and debugging. Your report must:

- explain the design and implementation of your file system using, as appropriate, diagrams for data structures and pseudo-code for algorithms;
- describe any additional features that you have chosen to implement;
- describe your testing strategy and any problems faced during development;
- discuss any limitations in your implementation and describe possible improvements

Approach

You must modify the 'myfs' implementation to meet the specified requirements. Only 'myfs.c' and 'myfs.h' should be modified.

Myfs is very simple and supports only a single directory, containing up to one file of up to 1000 bytes in size. Because there is only ever one directory there is need for only a single file control block that represents the root directory (struct myfcb the_root_fcb;). This global struct holds the meta-data for both the root directory and the file as well as the file name and identifier for the file data. Whenever any change to the meta-data is made, 'the_root_fcb' is written to the store.

Myfs uses a key-value store called 'unqlite' to store data persistently. Use of 'unqlite' is a key part of this practical exercise and all persistent data must be stored in the key-value store. Keys should be 128-bit uuids and there are examples of key generation in the code. Traversal of any file system starts with the root directory but it is necessary to know how to find the root directory. The starter code stores the file control block corresponding to the root directory using a fixed, predefined key, so it always knows how to find it. You should stick with this scheme but modify the root FCB to contain 128-bit uuids of any files contained inside the directory. These can then be used to fetch the FCBs corresponding to these files from the store.

The myfs implementation provides examples of how to fetch data from the store and how to write data to the store. You must ensure that changes to in memory data structures are written back to the store.

Your implementation must be much more flexible than the 'myfs' starter code and it is recommended that you implement something similar to the inode file systems described in lectures. Thus you must define a file control block (FCB) and use this to implement files and directories. A single bit of the 'mode' bits in the FCB distinguishes a directory from a file (see 'man 2 stat'). Directories must implement a mapping from names to the 128-bit uuids of the child objects (files or directories). The ***simplest suitable file control block*** is defined as follows:

```
struct fcb{
    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 change to meta-data (status) */
    off_t size;       /* size */
    uuid_t data;      /* data */
};
```

All of the fuse operations that you will need are implemented in 'myfs.c'. Your implementation will largely involve modifying the provided functions although you will also need to define additional structures and macros. At the very least ***you will need to define a structure for directory contents.***

There are many options when it comes to the implementation of file and directory data storage but here you are encouraged to keep is as simple as possible. While you could implement a hybrid indexing approach (as in inodes) you could also adopt a much simpler scheme and limit all files and directories to a single fixed or variable size data block. For instance, the simplest and most restrictive scheme would be to allocate a single 4kB block for each file or directory. Clearly this imposes a hard limit on file and directory size and in simple scheme the FCB would record the actual file size which may be less than the 4kB limit. A more flexible scheme would be to allocate variable sized blocks as required. The key-value store makes this second option as easy as the first. On the first write to a file a data block of some fixed size is allocated. If the file grows beyond this initial limit a new, larger, data block is allocated. The existing data would need to be copied to the new block and then the old block can be deleted and the new block written to the store (using the same key and thus avoiding the need to update the FCB). The problem with these simple allocation schemes is that all of the data for a file needs to be read into memory on every read or write.

During development and testing you will need a way to execute the same set of file system operations over and over again. It is recommended that you either write a set of short shell scripts or write some test programs to read and modify the file system in a controlled manner.

Debugging

If you want to debug your file system in gdb, you need to run the fuse file system in non-demonized mode. For this you will need two terminal windows. In one terminal:

```
$ gdb myfs
<set breakpoints>
(gdb) run -s -d <mountpoint>
```

In the second terminal run a program or issue shell commands to access/update the file system.

Resources

The following URLs contain examples and documentation that you may find useful:

- http://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201001/homework/fuse/fuse_doc.html
- http://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/
- https://github.com/libfuse/libfuse
- http://unqlite.org/

Note: You are encouraged to carry out our own background research on file system implementation. In your report you should clearly indicate any and all design decisions influenced by your research. Ensure that you cite your sources as appropriate.

Autochecking

This practical will not be autochecked.

Assessment

Only 'myfs.c' and 'myfs.h' will be considered during grading. 'myfs.c' and 'myfs.h' will be copied to a testing folder, compiled and run. Your code submission must not rely on additional source files or changes to the Makefile.

Marking will follow the guidelines given in the school student handbook (see link in next section). Some specific descriptors for this assignment are given below:

Mark range	**Descriptor**
1 - 6	A submission that does not compile or run, or perform any file operations.
7 - 10	A solution which implements some of the requirements and allows the user to do some basic operations on files, but corrupts regularly, or is too unstable for real use.
11 - 14	An implementation which is mostly correct, but has some significant problems. This could include missing important functionality, or serious issues with corruption after short use.
15 - 17	A well-written implementation which implements all required functionality, is usable in practice from the shell and other programs, and is accompanied by a clearly written report.
18 - 19	An excellent implementation which goes beyond the original specification by implementing one or more extensions such as indexing, caching, hardlinks with reference counting, renaming etc., and is accompanied by an excellent report.
20	An exceptional implementation showing independent reading and research, several extensions and excellent quality of code, accompanied by an exceptional, insightful report.

Policies and Guidelines

Marking
See the standard mark descriptors in the School Student Handbook:
http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Lateness penalty
The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):
http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good academic practice
The University policy on Good Academic Practice applies:
https://www.st-andrews.ac.uk/students/rules/academicpractice/