UNIX SYSTEM PROGRAMMING
LABORATORY PROJECT
(UE15CS352)

SIMPLE FILESYSTEM USING FUSE

Problem Statement: Implement simple file system using fuse library

Authors:

Abhuday Vibhanshu	(01FB15ECS011)
Aditya Jain	(01FB15ECS018)
Aniket Bharati	(01FB15ECS036)

About Fuse

Filesystem in Userspace(FUSE) is a software interface for Unix-like computer operating systems that lets users create their own file systems without editing kernel code. This is achieved by running file system code in user space while the FUSE module provides only a "bridge" to the actual kernel interfaces. The idea here is that if we can envision our interaction with an object in terms of a directory structure and filesystem operations, we can write a FUSE file system to provide that interaction. We will just have to write code that implements file operations like open(), read(), write(), etc. When our filesystem is mounted, programs are able to access the data using the standard file operation system calls, which call our code.

Addressing the project problem

Client programs can implement a userspace filesystem by providing a collection of functions that implement file system operations. The idea here is that if we can envision our interaction with an object in terms of a directory structure and filesystem operations, we can write a FUSE file system to provide that interaction. We will just have to write code that implements file operations like open(), read(), write(), etc. When our filesystem is mounted, programs are able to access the data using the standard file operation system calls, which call our code.

A FUSE filesystem is a program that listens on a socket for file operations to perform, and performs them. The FUSE library (libfuse) provides the communication with the socket, and passes the requests on to your code. It accomplishes this by using a "callback" mechanism. The callbacks are a set of functions we write to implement the file operations, and a struct fuse_operations containing pointers to them.

In the case of our filesystem, the callback struct is named as operations. There are a total of 12 file operations defined in our program with pointers in operations. Looking at a part of the initialization of the struct we see

```
static struct fuse_operations operations = {
    .getattr        = do_getattr,
    .readdir        = do_readdir,
    .read           = do_read,
    .write          = do_write,
    .truncate       = do_truncate,
    .mkdir          = do_mkdir,
    .rmdir          = do_rmdir,
    .create         = do_create,
    .unlink         = do_remove,
    .open           = do_open,
    .rename         = do_rename,
};
```

This indicates that operations.getattr points to do_getattr(), operations.readlink points to do_read(). Each of these functions is the re-implementation of the corresponding filesystem function: when a user program calls read(), my do_read() function ends up getting called. In general, what all of my implementations do is to log some information about the call, and then call the original system implementation of the operation on the underlying filesystem.

Implementation

To implement our functions we first created our internal data structures for file table and directory table. File table has entries for name of the file, its size, index values and dirty bit which is associated with a block of memory which indicates whether or not the corresponding block of memory has been modified. It also has an entry for file content which stores its data in the form of blocks. This is done to make our file system persistent so that we are able to shut down our file system (either through unmount or a machine reboot) and have all the files remain intact and in the same state when the file system is remounted. In the same way we implement our directory tables.

To implement our basic file I/O operations such as open, close, read, write, seek and directory functions such as mkdir, readdir etc. we write into our internal structures. Read(do_read) function creates an entry into the file table and when we remove our file (rm command) its entry gets removed from our file system. In the same way we implement mkdir and rmdir functions.

To implement a persistent file system we create blocks and then use a backup folder to store the name and the contents of the file. This is done for the directory as well, where we store the name and contents of the directory. So when we unmount the filesystem, we log the contents of our file system into our backup folder and then when we mount our file system we use the log files such that our file system becomes persistent.

REFERENCE AND CITATIONS

The following were taken a look for doing this project :-

[1] https://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/

[2] https://engineering.facile.it/blog/eng/write-filesystem-fuse/

[3] https://docs.racket-lang.org/fuse/index.html