CS-570 Advanced Operating Systems

Assignment 3: Log-Structured File System

Due: Saturday, 6th of May, 11:55 PM

This assignment has been divided into seven parts. Each subsequent part can be extended from the previous one.

Instructions

- You've to implement this assignment individually
- There will no extension of deadline. The assignment requires some time. Please start early.
- Absolutely no sharing of code is allowed even if it is a not so important helper function. Your code will be matched against other submissions and submissions from prior years. All plagiarism cases will be forwarded to Disciplinary Committee.

Goal

Understand and implement zero cost snapshots in a copy on write log structured file system using FUSE.

Simplifications

- Block size of the file system is 1K bytes.
- Your file system is not implemented over a block device like normal file systems. It is implemented on top of a file on an existing file system. You will treat this file as your backing store i.e. the block device. So block 20 will be the data in this file from $20 \times 1K$ to $20 \times 1K + 1023$.

FUSE

FUSE is a way to write a new file system in user space. All calls made by processes using the new file system go in the kernel and are forwarded into a user program. Of course the user program does not have direct access to disk drive blocks so the way FUSE file systems work is by storing their data in a normal file (provided by a regular kernel driver) or on a network server. This makes developing new file systems very fast. FUSE is available on Linux and Mac or you can install a Linux VM if you are on Windows. You do not need admin rights as you are not making any change in the kernel. FUSE API is available for many languages: C/C++, Java, Python etc. Find a FUSE tutorial for the language you are most comfortable with. Here are some useful resources.

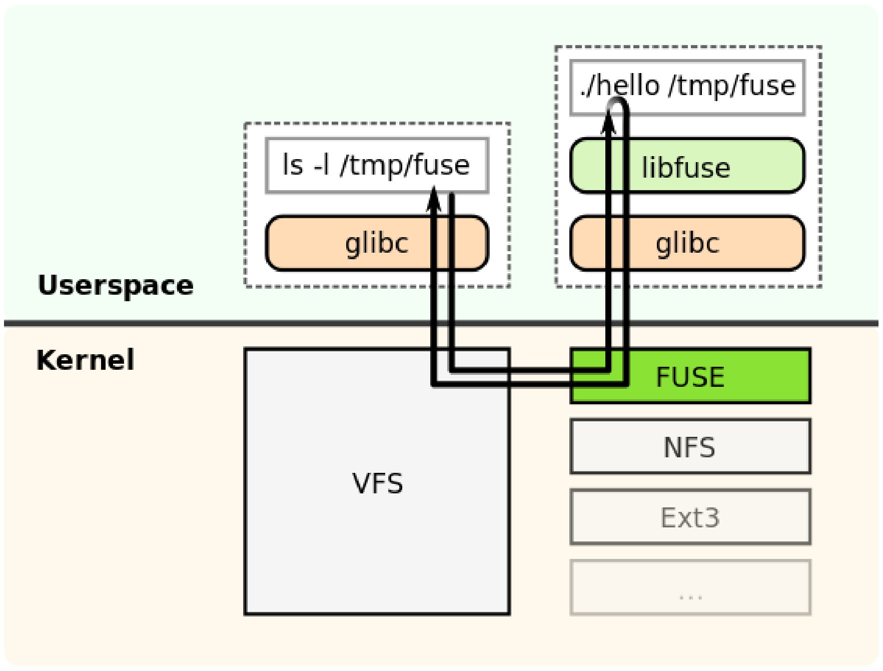

Figure 1:

- `http://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/fuse_doc.html`
- `https://stuff.mit.edu/iap/2009/fuse/fuse.ppt`

Required Interface

Your FUSE client `lfs_fuse` will be executed like this:

```
$ lfs_fuse -s -d <lfs_mount_dir> <lfs_filename>
```

`-s` signifies single-threaded operation, `-d` is debug foreground mode. `lfs_mount_dir` is an empty directory where your file system will become visible. These three options will be passed on to FUSE library. `lfs_filename` is the single file inside which you are storing the complete log using regular file I/O.

If `lfs_filename` does not exist, create a blank file system. Once `lfs_fuse` is running, the contents of our file system will be visible in `lfs_mount_dir`.

1: Reading and Writing Blocks (2 points)

Design a FUSE file system with a 1KB block size where if you read/write file 27 it means reading/writing whatever is in block 27. So when you mount your filesystem, you will see files equal to the number of blocks in the file system and each file will be exactly 1KB.

2: Reading and Writing Blocks through Inodes (2 points)

Now extend your file system such that the first 40 blocks contain 1024 inodes of 40 bytes each. Each indoe is 4 bytes of size followed by eight 4-byte block numbers and last 4 bytes can be ignored for this part. If you read/write file 27 it means accessing the file referred by inode 27. When you mount the filesystem youll now see 1024 files of 0 bytes and when you write to them, they can grow up to 8KB each.

3: Introducing a Directory (2 points)

Now define inode-0 as a special inode explaining what goes in the root directory. Inside the blocks referred to by this inode store inode and filename pairs separated by space with each entry on a new line.

At this time and in all later parts, your filesystem should start behaving like a normal filesystem but without any support for making new sub-directories. Your program should work for a normal workflow i.e. copying files to and from, editing directly with an editor, listing directory entries etc. Start putting an error message in all FUSE event handlers and when you run your file system, you will know which events are called when you do these actions.

4: Making data blocks log-structured (2 points)

Now make the data blocks log-structured i.e. you will **always** append the new data block and **never** change an existing data block. Once changed, you can update the inodes in its original place.

5: Making inode blocks log-structured (2 points)

Now make the inode blocks log-structured. You will need to somehow remember where is the most updated version of each of the 40 blocks of the inode table. Lets introduce a superblock in block-0 that stores this information. Now the superblock is the only block you can change in place. Everything else is appended at the end of the file. You do not need redundant superblocks as done in the paper. You are allowed to overwrite the superblock after every write (even though this defeats the whole purpose of avoiding seeks).

6: Indirect blocks (2 points)

Use the last 4 bytes in the inode to store an indirect block pointer if the file grows beyond direct blocks. After this step, your file system should support a maximum file size of 264KB.

7: Checkpoints (3 points)

Make sure your data and inode blocks are completely log-structured (no over-writing) or else this part is almost impossible to do.

There should be a read-only directory `checkpoints` in the root folder. Inside it there should be one directory for every checkpoint taken so far. If an empty file system is mounted at `/tmp/lfs`, then this is how you create a new checkpoint:

```
$ ls -a /tmp/lfs
.  ..  checkpoints
$ ls -a /tmp/lfs/checkpoints
.  ..  checkpoint
$ cat /tmp/lfs/checkpoints/checkpoint
15
$ ls -a /tmp/lfs/checkpoints
.  ..  checkpoint 15
$ ls -a /tmp/lfs/checkpoints/15
.  ..
```

To create a new checkpoint, you take the current superblock and write it as a new data block at the end of the log and return its block number to the user. You also need to store this in the file system so all previous checkpoints can be shown in the `checkpoints` special folder. One way to do this is to write the block number of last checkpoint in the super block. This way, all checkpoints will be chained together. Creating a checkpoint should take exactly 1KB i.e. one additional block.

When `readdir` is called on the `checkpoints` directory you follow this chain and show all checkpoint block numbers as directories in addition to the special file `checkpoint`. When a read request comes for a file with a path name starting with `/checkpoints/X` consider `X` the block number of a checkpoint and lookup the remaining path in the usual manner but using the checkpoint block instead of the superblock. Once you have the alternate checkpoint block, you should be able to use the same code as for regular reads. Writes to files in old checkpoints should not be allowed.

Have Fun☺