Assignment 2: Log structured file system

Deadline: `Sunday 19th April 23:59`

Logistics

- You can do it alone for all 10 marks or in a group of two for a maximum of 8 marks. This is to encourage good programmers to do it alone and the rest to pair up and still put in a good effort. No request for a group of three will be allowed.
- 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 Diciplinary Committee.

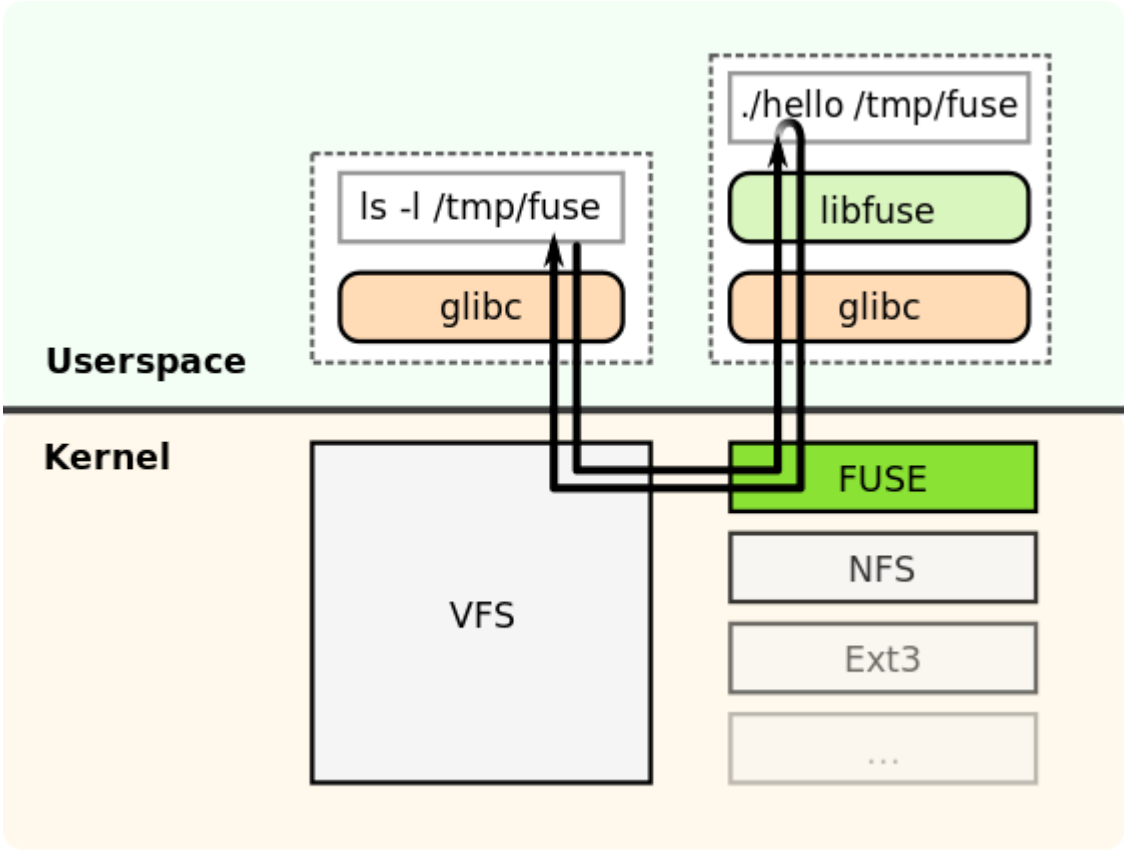
Goal

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 20x1K to 20x1K+1023.
- You will not implement any cleaning. Deleted blocks are wasted.

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 (http://fuse.sourceforge.net) and Mac (http://osxfuse.github.io) 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.

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

Reading and Writing Files

Design an inode based file system with a 1KB block size in a log i.e. you will **always** append to the log file and **never** change an existing block except the very first block which will the be superblock containing information about which blocks contain the latest inode table. You do not need two superblocks as done in the paper.

Its a good idea to implement `readBlock` and `writeBlock` methods that take a block number as argument. And similarly `readInode` and `writeInode` methods which read a given entry from a given inode number using a given superblock.

You are not required to support creating new directories. However the root directory with a fixed inode number should be stored as a regular file containing filename and inode number pairs. It might be a good idea to have a hard-coded directory at first (e.g. file1->inode1, file2->inode2, file3->inode3) and when you can successfully read/write these files, then implement creating and deleting files.

You should support a 1000 files at least (i.e. 1000 inodes). Maximum file size supported should be at least 30 KB. A block number should be at least a 4 byte integer. Evaluate if you need indirect pointers in i-node. You are allowed to overwrite the superblock after every write (even though this defeats the whole purpose of avoiding seeks). The numbers in this paragraph are minimum requirements to simplify your task. You are welcome to support larger and more files and periodic superblock update instead of immediate.

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.

Checkpoints

Read this once you are done with the above section.

There should be a read-only directory `checkpoints` in the root folder. Inside it there should be a directory for every checkpoint taken so far and a single file named `checkpoint`. Reading from the special `checkpoint` file creates a new checkpoint. This is the easiest to implement since the call will come back in the same user program that is implementing the whole file system. You will have a special check for this filename and create a new checkpoint and send back the checkpoint number.

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.