OS 2014 - EX5
A Fuse File System

Supervisor - Noga H. Rotman

Due Date: 5/6/2014

Before you start, don't forget to read the course guidelines!
No extensions will be given without a good reason (Miluim, sickness or childbirth).

Assignment

In this exercise you will implement a simple **single threaded** caching file system using FUSE, and a simple algorithm to manage the file system's cache memory.

Caching file systems saves files or parts of them to a memory segment called cached memory, which is stored in the main memory. That way, if the user wishes to access information that is saved in the cache, it can be retrieved from the cache instead of the disk, thus reducing the number of disk access required.

In order to make this system efficient, we want to ensure that the information that will be requested the most is saved to the cache, thus ensuring the disk will be accessed as seldom as possible. There are many complicated algorithms that deal with this problem - choosing the best one is an art all to itself; in this exercise you will implement a LRU memory management algorithm.

File System

Your file system will be implemented through a file called MyCachingFileSystem.cpp, and should be accessed using the following command:

```
$ myCachingFileSystem rootdir mountdir numberOfBlocks blockSize
```

Where *mountdir* is the mount point directory - which should be empty, and *rootdir* is a folder containing files, such that once your file system is mounted, *mountdir* should respond as if it contains all the files in *rootdir*: the content should be the same, and the user should be able to open and read those files (through your filesystem of course). All the actions done on a file in *mountdir* should actually be done on the corresponding file in *rootdir* (renaming for example).

The cache will be made out of *numberOfBlocks*, where each block is made out of *blockSize* bytes (you can assume you'll not be asked to read a file bigger than the cache size). Each block pertains to a specific file; each file is divided into blocks of *blockSize* bytes, each block starting byte is always a multiplicity of *blockSize*.

When information is retrieved from the disk, it will be saved into a cache memory, according to the following rule:

- If the file is smaller than the block size, it will be saved in its' entirety to the cache.
- Otherwise, only the requested block/s will be retrieved from the disk and saved to the cache.

For example, say *blockSize* is 5 bytes, and given a file of 17 bytes, we wish to read bytes 13-15. Then block 3 should be read and entered into the cache (if it wasn't there previously).

Upon access to a file, if the requested block is already in the cache, the information will be retrieved from it. Otherwise, the information will be brought from the disk, and saved to the cache, as described above.

To do so, your FUSE file system implementation must override following functions (note that some of these functions may have an empty implementation):

- getattr
- access
- open - should ~~always open a file with the O_DIRECT and O_SYNC flags~~. The max path length is PATH_MAX that can be found in limits.h.
- read
- fgetattr
- flush
- release
- opendir
- readdir
- releasedir
- rename
- init
- destroy

- ioctl - this function will write the current status of the cache to a file named "ioctloutput.log", which will be positioned in the current working directory. If the file already exist, the information will be appended to the existing file in a new line. The information will be formatted as follows:
  first line: the time when the icotl command was called *(use gettimeofday as you did in the previous exercises 1 and 2)* in the format month:day:hour:minutes:seconds:miliseconds in local time.
  Next will be a line for every block in the cache, that will list (in the following order):
    - The name of the file - relative path from root/mountdir.
    - Number of the block (not the number in the cache, but the enumeration within the file itself - starting with 1).
    - The time in the block was last accessed (in the format specified above)

  each element separated from the rest by a tab, in descending order of time the block was last accessed. For example:
  SomeFile.txt 2 04:02:06:00:00:422
  myFolder/SomeOtherFile.txt 1 01:01:23:14:22:083

Cache Management

You will implement a LRU algorithm to manage your cache memory:

- When saving information to the cache, if there is no room available, the block that was least recently used should be removed to make way for a new block.

Error Messages

Message Error	**Print to**	**Error Description**
"system error: couldn't open ioctloutput.log file\n"	stderr	When the log file can't be opened.
"system error: couldn't write to ioctloutput.log file\n"	stderr	When failing to write to the log.
"error in "+ name of your function +"\n"	ioctloutput.log	When an operation (system call or otherwise) that is imperative to your implementation fails.
"usage: MyCachingFileSystem rootdir mountdir numberOfBlocks blockSize\n"	stdout	When a user is trying to start the program and the parameters are illegal. **If the parameters are illegal, the program should exit, and fuse shouldn't be invoked in the first place.**

Assumptions, Errors, etc.

- As mentioned above, illegal parameters should cause the program to exit. Illegal parameters are those who won't let the program work as it's expected to, such as one of the directories does not exist, the number of blocks / block size is 0 or smaller, etc.
- You can assume that in case the mountdir exists, it is empty.
- You can also assume we'll not be supplying a cache size (block size * number of blocks) that's too big.
- Once a file has been renamed, the cache should be updated accordingly.

Guidelines and Tips

- You may also use the fuse -s flag to make the system single threaded.
- Use the -f flag when running FUSE to bring the program to the foreground - this means you'll be able to see the cout, but also means you'll need a second shell to access your folder
- We've provided you with a basic file you can start working on. Implement all the functions, make sure the rootdir and mountdir are correct (the mountdir should be the second argument passed to the fuse main function) and then start testing. If you choose not to use it, be sure to define the FUSE version prior to including the fuse.h
- Global variables should be handled through fuse_get_context()->private_data - see the fuse.h documentation and bbfs system for more details.
- ~~Because of the O_DIRECT flag, you'll need to use the posix_memalign function on the buf before the actual read. One example of how to use it can be found here.~~
- The mountdir and rootdir should be in the /tmp folder (do **not** use the same folder).
- Note that the above only relates to **reading** data, not writing data - which means you don't have to implement anything that has to do with writing.
- **Do not run the exercise on river!** When there's a problem with fuse it tends to get stuck, and you won't be able to keep on working. You may also be hurting your friends, so we will be deducting points if you choose to ignore this instruction.
- When encountering a problem on the lab computers, restart the machine.
- As usual, your code should not have any memory leaks.
- Use the Q&A forum, and be sure to check the existing topics, as some of your questions might have already been answered.
- Make your code readable (indentation, function names, etc.).

Background Reading and Resources

1. FUSE API documentation
2. Writing a FUSE Filesystem is a good tutorial.

Submission

Submit a tar file on-line containing the following:

- A README file. The README should be structured according to the course guidelines, and contain an explanation on how and why your library functions are built the way they are.
- MyCachingFileSystem.cpp, containing your implementation of the file system, and all other relevant files you implemented.
- Your Makefile.