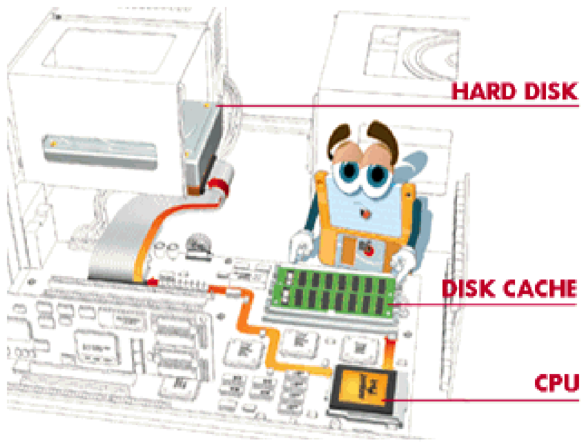
A Caching File System

OS 2016 Exercise 4

Supervisor – Netanel Zakay

Caching Meaning

Many times the term cache refers to the CPU cache, which is used by the central processing unit (CPU) of a computer to reduce the average time to access data from the main memory. The CPU cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations.



However, the word *cache* has wider implication in computing. A cache is a component that stores data so future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation, or the duplicate of data stored elsewhere. A cache hit occurs when the requested data can be found in a cache, while a cache miss occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store; thus, the more requests can be served from the cache, the faster the system performs.

To be cost-effective and to enable efficient use of data, caches are relatively small. Nevertheless, caches have proven themselves in many areas of computing because access patterns in typical computer applications exhibit the locality of reference. Moreover, access patterns exhibit temporal locality if data is requested again that has been recently requested already, while spatial locality refers to requests for data physically stored close to data that has been already requested.

For example, we can cache data located in the memory by saving it in the CPU cache, cache files located in the disk by saving them in the memory (this is called buffer cache or disk cache), or cache HTTP pages located on a remote server by saving them in a router of the Internet Service Provider [ISP] (this is called web cache or HTTP cache). In this exercise we will focus on the buffer cache,which caches files in the memory.

Caching Algorithms

Cache replacement algorithms are algorithms that a computer program or hardware component can use to manage a cache of information stored on the computer. When the cache is full and new information arrives, the algorithm must choose which items to discard to make room for the new ones.

There are several common algorithms to choose which item to discard:

- Least Recently Used (LRU) discards the least recently used item first. This is one of the most used algorithms.
- Most Recently Used (MRU) discard the most recently used item first (opposite of LRU).
- Random Replacement (RU) randomly selects a candidate item and discards it.
- Least Frequently Used (LFU**)** counts how often each item is used, and discards the least frequently used one first.
- Least Frequently Used Aging is a common variant of LFU in order to solve a major problem: a formerly popular item becomes unpopular remains in the cache for long (unlimited) time, preventing new items from replacing it.

LRU and variations of LFU are very popular approaches. Several works tried to combine them into a single algorithm. IBM's Adaptive Replacement Cache (ARC) is an interesting example for that. An alternative approach is the Frequency-Based Replacement (FBR) that you will implement in this exercise.

FBR is a hybrid replacement policy, attempting to capture the benefits of both LRU and LFU without associated drawbacks. FBR maintains the LRU ordering of all blocks in the cache, but the replacement decision is primarily based upon the frequency count. In other words, the data structure is sorted by last access time. To accomplish this, FBR divides the cache into three partitions: a new partition, a middle partition, and an old partition. The new partition contains the most recent used blocks

(MRU) and the old partition the least recent used blocks (LRU). The middle section consists of those blocks not in either the new or the old section. The size of each partition is one of the inputs of your program.

When a new block is added to the cache, its references count is 1. In any reference to a block, it should be placed on top of the stack, as it is accessed now. When a reference occurs to a block in the new section, its reference count is **not** incremented. References to the middle and old sections do cause the reference count to be incremented. When a block must be chosen for replacement, FBR chooses the block with the smallest reference count from the **old partition**, choosing the least recently used such block if there is more than one.

For furthermore clarifications and motivations, you **must** read sections 2.1 and 2.2 in here (this link is available within HUJI). It's only one page that explains how FBR works and why. Pay attention – section 2.3 and so on are not relevant for this exercise.

Background Reading and Resources

1. Read and understand the relevant TA class is a first step
2. FUSE API documentation
3. The Big Brother File System (later will be referred as "bbfs") is a good tutorial.

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 buffer cache. This file system will actually become part of the file system you see and can use on your machine, but you'll be responsible for how it works!

Caching file systems saves files or parts of them to a memory segment called the buffer cache, 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 memory instead of the disk, thus reducing the number of disk accesses 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. Therefore, we will use the LFU algorithm to choose which item to discard from the cache when we need more space.

Managing a complete cached file system is an interesting task, but away too big for an exercise in our course. Therefore, the files in our file system will be read-only (except rename), and you may assume that the files won't be changed at all after mounting your file system. Moreover, while in reality we would cache all the files' inode data (including the stat, size, etc), here you are requested to cache only the content of the file (e.g. the data received from "read" request).

To manage a caching file system, you will need two relevant variables – the number of blocks (*numberOfBlocks*) to save in the cache, which is an input of the simulation, and the size of each block (*blockSize*) which depends on the OS (see more details later). This enables you to cache *numberOfBlocks***blockSize* bytes. You may assume that this value is not too big (so you can allocate it on the heap without problems). The blocks in the cache must be aligned according to the *blockSize*. This means that when you retrieve file content from the disk (following a read request), you will always be asked for complete blocks with an offset that is a multiple of *blockSize*.

Let's demonstrate the behavior of such a system, when *numberOfBlocks* is 10 and *blockSize* is 1024. We start with an empty buffer cache. Assume that we also have a file named "tmp" with 1500 bytes in it.

1. If a user tries to read the last 500 bytes, we must retrieve all the file (two blocks) and save both its blocks in the cache (we can't read only the required 24 bytes from the first block because we can't read partial blocks).
2. If a user tries to read the last 50 bytes of the file, we will retrieve from the disk and cache the second block of the file (bytes 1024-1499). However, we will return only its last 50 bytes to the user (bytes 1450-1499).
3. If a user does 2 and then tries to read all the file:
   In the first read, the file system retrieves and caches the second block, returning only its last 50 bytes. In the second read, the file system retrieves and caches only the first block, because the second block is already in the buffer cache and shouldn't need to be read again. Of course, the function will return all the file.

You may see from this example that the function always returns to the user all the information he requested, but retrieves from the disk only the blocks that are not already in the buffer cache.

Running Your File System

Your file system will be implemented in a file called CachingFileSystem.cpp, and should be activated using the following command:

$ CachingFileSystem *rootdir mountdir numberOfBlocks fOld fNew*

Where:

- *mountdir* is the mount point directory - which should be empty. The files in your file system will appear to be in this directory.
- *rootdir* is a directory (folder) containing files, which will be mounted by your filesystem at *mountdir*. This means that when using your file system, *mountdir* should respond as if it contains all the files in rootdir, and *rootdir* is where they really are. Thus the user should be able to open and read the files in *rootdir* (through your filesystem of course) using a path through *mountdir*. All the actions done on a file in *mountdir* (again, using your file system) should actually be done on the corresponding file in *rootdir* (renaming for example).
- *NumberOfBlocks* is the number of blocks in the buffer cache
- *BlockSize* is the number of bytes in each block.
- fOld is the percentage of blocks in the old partition (rounding down)
- fNew is the percentage of blocks in the new partition (rounding down)

For example: CachingFileSystem /tmp/myRootDir /tmp/myMountDir 100 0.3333 0.5
Creates a caching file system with 100 blocks with 33 blocks in the old partition, 50 in the new partition, and the remaining 17 are in the middle partition.

Supported Functions

Your FUSE file system implementation must override the following functions (note that some of these functions may have an empty implementation):

- *getattr*
- *access*
- *open*
- *read*
- *fgetattr*
- *flush*
- *release*
- *opendir*
- *readdir*
- *releasedir*
- *rename*
- *init*
- *destroy*
- *ioctl*

The functions are described in the supplied CachingFileSystem.cpp file. While you need to implement several functions, in many of them the main task of your function is only to find an equivalent function in Linux, and to call to this function. For example, the *getattr* function initializes the stat structure with the file attributes. Pay attention that in your file system, the files (in *mountdir*) have exactly the same attributes as the real files in the root dir. Therefore, you only need to initialize the stat of the requested file in the root directory.

However, there are several functions that are a bit more complicated and we will describe them here.

- ***open***. While open is a simple function, pay attention that you should use the given flags (in the *fuse_file_info*).
- ***readdir***. This function implementation is a bit more complicated than a simple forward. Pay attention to the difference between what you are requested to do and what Linux's "*readdir*" does.
- ***read***. The read function is the most important function in your file system. It is actually the only function that uses your cache. Most of the logic of the program, as described previously, should be in here.
- ***rename***. The rename function renames a file. Pay attention that this file may have cached blocks in the file system's buffer cache. In this situation, the cache must be updated such that when there is an open and read of the renamed file, the information that was in the cache before the "rename" should be identified and should not be retrieved from the disk again. Of course, the rename function doesn't change the accesses counter to each block.

- ***ioctl***. This function will write the current status of the cache to the log file (see details later). The function prints a line for every block that was used in the cache (meaning, with at least one access). Each line contains the following values separated by a single space.
  - The name of the file - relative path from *mountdir*.
  - Number of the block (not the number in the cache, but the enumeration within the file itself - starting with 1).
  - The reference count described above (pay attention – this is a positive value).

  The order of the entries is from the least recent used to the most recent used.

  For example, the following lines could be appended to the log file after iocl is called:
  1430334011 ioctl (see details about this line, later)
  SomeFile.txt 2 1
  myFolder/SomeOtherFile.txt 1 5

O_DIRECT and O_SYNC Flags

O_DIRECT and O_SYNC are optional flags of the open command. You **must** use the version of "open" that receives two parameters, and use the following flags as the second parameter: **O_RDONLY | O_DIRECT | O_SYNC**. Please copy and paste these flags because it will be checked automatically by script. Let's try to understand what these flags do and why we need them.

O_SYNC is the more simple flag. This flag is used for synchronous I/O. For example, any write on the resulting file descriptor will block the calling process until the data has been physically written to the underlying hardware.

O_DIRECT tries to minimize cache effects of the I/O to and from this file. While in regular files we want the OS to use the buffer cache in order to improve performance, here we want to manage the cache in our virtual file system and therefore we want to access the disk directly without the OS caching (to avoid double caching of the same blocks). File I/O is done directly to/from user-space buffers. The O_DIRECT flag on its own makes an effort to transfer data synchronously, but does not give the guarantees of the O_SYNC flag that data and necessary metadata are transferred. To guarantee synchronous I/O, O_SYNC must be used in addition to O_DIRECT.

Using these flags mean that you will access the disk directly. However, the disk can transfer data only in blocks (fixed size). The following lines return the block size in your OS:

```
struct stat fi;
stat("/tmp", &fi);
int blksize=fi.st_blksize;
```

Accessing the disk directly leads to a few restrictions for the read/pread commands:
1. You must read exactly a single block in each time.
2. The start address (in the file) must be aligned with the block size.
3. The address of the buffer must be aligned with the block size.
   For this purpose, you may use the function *aligned_alloc* which allocated memory with alignment.

If any of these restrictions is violated the read function fails (and returns negative value). For example, if the block size is 512, and the address of the buffer that I is not aligned to 512, the read fails.

Logging Your File System Behavior

Your program will log its activity to a hidden log file named "*.filesystem.log*". This file will be positioned under *rootdir*. You need to create the log in the main function, before invoking FUSE. If the file already exists, then append to it and don't recreate (empty) it.

The log should contain exactly the following information (without anything else). In each file system function that you implement (described above), when the function is called you need to write to the log: "UNIX_TIME FUNCTION_NAME" where FUNCTION_NAME is the name of the function and the UNIX_TIME is the POSIX/Epoch time (use "time" function to obtain it). For example, when a user uses "*open*", our "*caching_open*" will be called. Our first step within the function is to write "1430374011 open" (and not *caching_open*!) to the log. Pay attention that you don't log your private functions, but only the file system functions. You should check before submitting your code that you have exactly 14 such print instructions in the code (one for each function). In addition to that, the log also contains the FBR information printed by the *ioctl* function as it was described above.

The *.filesystem.log* is designed to help us improve or check the system. The users that use your system (via the mounted directory) shouldn't be aware of the existence of this file. To do that, you must block them from using this file in any of your functions. For example, an attempt to open the log file will fail and return an error. Specifically, any attempt to access the log file should return -*ENOENT*, which is an error defined in *errno.h* meaning "no such file or directory". We will speak more about errors and the functions' return values later.

Pay attention that in order to cause the users not see the file via the *ls* command, it's not enough to block an access to the .filesystem.log. Think why it happens, and how you can solve this problem.

Note – while we previously said that *mountdir* should reflect the content of the *rootdir*, the log is an exception. It exists in the *rootdir* but is invisible using *mountdir*. This simple example demonstrates that the files that we see are not the real data on the disk. In this way, it is possible to create hidden files (e.g. all the files started with "." in Linux).

Errors handling in the main function

When your program starts, you first must check the received parameters in main. If the number of parameters is wrong, or the parameters values are invalid, you must print (to *stdout*) the following message:
"Usage: CachingFileSystem rootdir mountdir numberOfBlocks fOld fNew\n".

Invalid values are:

- The *rootdir* and the *mountdir* are invalid if the directories don't exist.
- The *numberOfBlocks* is invalid if they are not positive numbers (zero is invalid too).
- fOld is invalid if it is not a number between 0 to 1 or if the size of the partition of the old blocks is not positive.
- fNew is invalid if it is not a number between 0 to 1 or if the size of the partition of the new blocks is not positive.
- Also, fOld and fNew are invalid if the fOld+fNew is bigger than 1.

If there was another error in the main function (e.g. failure in alloc), you must write to the stderr a message starting with "System Error: " and exit.

In any error in the main function, the program should exit, and FUSE shouldn't be invoked in the first place (this means that after running FUSE's main, you shouldn't make any system call in the main function [except free if you need], because they may fail after invoking FUSE**).**

Errors in the File System Functions and Their Return Values

Each system call and library function may fail. When they fail, they update the *errno* value, which contains the last error number. *errno.h* is a library file, which defines an int value for each possible reason for an error.

When you use these functions in your file system, you must check the return value of each such call in order to understand if an error occurred. If there was an error, also your function must fail. In this case, you must return "*–errno*" (you need the minus before the *errno*, because *errno.h* defines positive values, and a minus value indicates an error). Pay attention, if you won't preserve this behavior and return the correct error, commands like "mv" in the shell will fail (you are welcome to try it yourself!).

You are required to stick with this behavior for each failure in your file system functions. For example, a failure to open the log file, to print into the log file, *malloc* failure, etc. However, there is a single failure that is not related to external function – when a function is requested to access the log file. In this case, return a -*ENOENT* error (no such file or directory).

caching_init and *caching_destroy* are the only two functions that don't return int, and therefore can't return error. If failure occurs during these functions, do nothing (absorb the failure and don't report it). Don't return the error value from the init function. For your task, the function needs to return NULL always (if you do something else, be sure to use the *fuse_context* correctly).

Assumptions

- You can assume that in case the *mountdir* exists, it is empty, and it's not within the *rootdir*.
- You can also assume we'll not be supplying a cache size (block size * number of blocks) that's too big.
- The files won't be changed after running your system (remember that the requested file-system commands doesn't change the files' content).
- You may assume that we won't use "links", so each file will be defined by a single path.

Guidelines and Tips

This part is extremely important. Read the tips carefully, they might help you and save your precious time.
If you are stuck or unsure what to do, read the tips again and make sure that you follow them.

- ***CachingFileSystem*.cpp**. We've provided you with a basic file that you should start working on. Your goal is to implement all the functions in this file.

- ***fuse_main* arguments**. This function may get a few arguments (using its first two parameters). In the *CachingFileSystem* that we provided you with, we used the "*-s*" flag, which makes FUSE run in a single thread. It's extremely recommended that you won't change it (to avoid multi-thread problems).
  Another important flag is "*-f*". It is used when running FUSE to bring the program to the foreground - this means you'll be able to see the cout (for debugging purpose), but also means you'll need a second shell to access your folder. **Don't forget to remove the -f flag before submitting your exercise.**

- **fuse_file_info**. The *struct fuse_file_info *fi* that you receive in many functions contains several fields. However, there are only two fields that are relevant for you. The first is the "flags" which must be used in the "*open*" command. The other field is "*fh*", which you are responsible to init, delete and use in multiple situations.

- **Global variables**. In FUSE, global variables should be handled through *fuse_get_context()->private_data* - see the fuse.h documentation and bbfs system (in the background) for more details.
  That being said, global **static** variables are valid and work as expected. While it is considered as bad design in applications, globals are widely used in operating systems, and for our purpose they are enough and it will help you to avoid unnecessary complications.

- To compile your code in g++, use this line.

- ***mountdir* and *rootdir* folders**. The file system for your user name is the Network File System (NFS). Therefore, the *mountdir* and *rootdir* can't be located there. Instead, **you must use directories for the *mountdir* and *rootdir* under the */tmp* directory.**
  Moreover, **don't use the same folder for mountdir and rootdir, and make sure that mountdir is empty**.

- **Running FUSE**. FUSE changes the file system, which may lead to very strange behaviors, including crashing the operating system if you do something incorrectly. When encountering a problem on the lab computers, restart the machine.
  The systems' servers (e.g. river) don't support FUSE. This is a lesson from last year, when River collapsed multiple times. We strongly suggest you to work from the Aquarium or Rotbergs' open space. It is possible to run your code at home using a Linux virtual machine. In this way, a bug won't affect your operating system, but only the virtual machine, and a restart won't be needed. However, you may then need to make some adaptations to run your code here (we had such cases last year).

- Relating to reading with O_DIRECT flag. First, I highly recommend you to create a basic program without FUSE that opens a file using the required flags and read from the file. It is very simple, and after a basic experience you will understand the restrictions better. You may also start without using O_DIRECT (however, don't forget it before submitting your final solution!). These tips will lead to a separation between problems in FUSE and problems arising from the O_DIRECT flag.

- **Code Design**. As usual, make your code readable and as simple as possible, and you shouldn't have any memory leaks. In this exercise especially I would suggest to invest in a good design. Keep the logic of the FBR (and maybe the calculations of the blocks to bring) in separate files.

- **Working step by step**. Don't write all the code and then try to run it! Write a single file-system function and check it, only when it works move on to the next one.

- **Understand each single line in your code**. We supplied you with a file containing empty functions in purpose. This is meant to encourage you to start with a clean slate. We also provided you with references to two FUSE projects: the hello world from the class's presentation and the bbfs. Use them in order to understand what your functions need to do, and how to work with FUSE's structs.
  We strongly advise you to minimize the copy-and-paste from these examples and use them mainly for learning. In any case, under any circumstances, don't use functions that you don't understand. This will lead you to have bugs that you don't know how to solve because you don't understand your own program. If you do such things, neither we nor your friends will be able to help you.

- In contrast to other exercises that you did (in the university and in our course), here you use an interface with a low-level, complicated library. Be careful, understand what you need to do and what your code does. You will need to google to understand the Linux functions. We probably won't be able to solve each problem you encounter, because we didn't have the same problems. Therefore, we strongly encourage you to help each other and reply to your friends' posts in the Q&A forum.

- **Unmounting your file system**. Pay attention that when you run your program, you mount the file system. Therefore, you must unmount it (use "*fusermount -u*") before running your program again.

- After running your file system, its working directory may be changed to "/" (it happens when you don't use the "*f*" flag). If you need the working directory for some reason, you should save when the main is called (possible as a global static char*).

- **Testing your program**. We suggest you to do the following two steps in order to test your program.

  - **Basic test using shell commands.** Pay attention that the commands that you implement aren't equivalent to the common shell commands. However, the shell commands use your functions. You can *cd* into directories, *ls* to see the files, open and read the files by *cat* or *tail* (you may use the *–c* flag), and *mv* files within the mount directory (which uses your rename). However, be aware that each shell function may use multiple functions of yours. For example, if "mv" doesn't work, it doesn't necessarily mean that you have a bug in rename. The source of the bug may be in other function that "mv" uses.
    Pay attention that "*ioctl*" can't be called in this way. In order to use this function you may copy and paste the following line to the shell:
    ```
    python -c "import os,fcntl; fd = os.open('/tmp/fuse_mount', os.O_RDONLY); fcntl.ioctl(fd, 0); os.close(fd)"
    ```
    where "/tmp/fuse_mount" is the mount point of your file system.

  - **More sophisticated tests** should be done via scripts. There are things that are harder (but still possible) to check via the shell, such as the caching algorithm. To test your code properly, we highly suggest you to create testers. These testers may check automatically that the log file can't be accessed, that you always preserve the correct information in the log file (by reading specific blocks from files), that you manage the cache properly and that the rename function works as expected.

Theoretical Part (10 points)

Don't write more than a few lines per question.

1. In this exercise you cached files' blocks in the heap in order to enable fast access to this data.
   Does this always provides faster response then accessing the disk?
   Hint: where is the cache saved?
2. In the class you saw different algorithms for choosing which memory page will be moved to the disk. The most common approach is the clock-algorithm, which is LRU-like. Also our blocks-caching algorithms tries to minimize the accesses to disks by saving data in the memory. However, when we manage the buffer cache, we may actually use more sophisticated algorithms (such as FBR), which will be much harder to manage for swapping pages. Why?
   Hint – who handles accesses to memory? And who handles accesses to files?
3. Give a simple working pattern with files when LRU is better than LFU and another working pattern when LFU is better. Finally, give a working pattern when both of them don't help at all.
4. In FBR, accesses to blocks in the "new section" doesn't increase the block's counter. Why? Which possible issues it tries to solve?

Submission

Submit a tar file on-line containing the following:

- A README file. The README should be structured according to the course guidelines, and contains an explanation on how and why your library functions are built the way they are, as well as answers to the theoretical questions.
- CachingFileSystem.cpp, containing your implementation of the file system, and all other relevant files you implemented.
- Your *Makefile* with all the requested commands.
  The *make* command should create an executable named *CachingFileSystem* .

Late submission policy						
Submission time	29.5, 23:55	30.5, 23:55	31.5, 23:55	1.6, 23:55	2.6, 12:00	3.6, 12:00
Penalty	0	-3	-10	-25	-40	**Course failure**