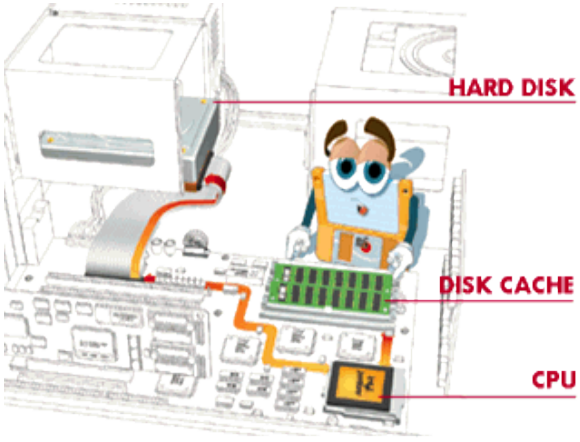
A Cache File System

OS 2016 Exercise 4

Supervisor – Netanel Zakay

Cache 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.

Cache Algorithms

Cache algorithms (also frequently called cache replacement algorithms or cache replacement policies) are optimizing instructions - or algorithms - that a computer program or a hardware-maintained structure can follow in order to manage a cache of information stored on the computer. When the cache is full, the algorithm must choose which items to discard to make room for the new ones.

The cache algorithms' goa is to minimize the number of cache misses. Ideally, we would like to discard the information that will not be needed for the longest time in the future (Bélády's Algorithm). However, it's usually impossible to predict how far in the future information will be needed. Therefore, there are several common practical approaches for cache algorithms:

- 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 (if multiple items have the same lowest frequency, any of them can be discards).
- 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.

In this exercise you will implement three different cache algorithms: LRU, LFU and FBR. 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 Frequency-Based Replacement (FBR).

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.

Assignment

In this exercise you will implement a library that represents a cache file system called ***CacheFS***. Using your library, users will be able to access files stored in the disk using a cache algorithms that they choose. The library support three cache algorithms: LRU, LFU and FBR.

Cache file systems saves files or parts of them to a memory segment called *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 improve the system's performance, the cache file system tries is to minimize cache misses, which will lead to less disk accesses. Due to the inability to implement the best algorithm (Bélády's Algorithm) and the fact that each algorithm is better under certain conditions, our CacheFS library provides three common cache algorithms. Users may use CacheFS to compare the number of disk access with each cache algorithm, and choose the best one for their purpose.

Managing a complete cache file system is an interesting task, but away too complicated for an exercise in our course. Therefore, our library supports only the read operation and you may assume that files won't be changed at all after opening them. 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 files (meaning - the data received from "read" request).

To manage a cache file system, you need the number of blocks (*numberOfBlocks*) to save in the cache, which is an input of the library, and the size of each block (*blockSize*), which depends on the OS (see more details later). This allows 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 ask 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 there is a file named "myFile" 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 add both blocks to 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.

As demonstrated by the example above, the *read* function always returns all the requested information, but retrieves from the disk only the blocks that are not already in the buffer cache.

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 cache (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 we use is not aligned to 512, the read fails.

Guidelines and Tips

- **Cache file system API.** The API of the cache file system is defined in the header: *CacheFS.h*. This header contains the signature and a full description of all the functions that your library supports (and therefore you need to implement). In a few words, the functions are:
  - *CacheFS_init* - initializes the library (allocates resources)
  - *CacheFS _destroy* - destroys the library (deallocate resources)
  - *CacheFS _open* - opens a file
  - *CacheFS _close* - closes a file
  - *CacheFS _pread* - reads data from a file
  - *CacheFS _print_cache* - prints the state of the cache

- **Assumptions**:
  - The cache size (*numberOfBlocks*blockSize*) is not too big and can be allocated and stored in the memory.
  - After using your library to open a file (using *CacheFS_open)*, we won't change the file at all. For example, we won't rename it, move it, delete it or change its content.
  - There is no concurrent use in your library. Only a single thread uses it. Therefore, there is no need to protect shared resources.
  - See more assumptions in the description of *CacheFS_destroy* and *CacheFS_init* in the API (*CacheFS.h)*

- **Error handling**. As described in the API, all the functions return negative number in case of an error (we don't care what value as long as it's negative). In any case, if the user follows the assumptions (mentioned above), your library should not collapse. This means that each system call or library function that you use must be checked.

- **Use only */tmp* folder.** The file system that is used in the Aquarium is Network File System (NFS). In this file system, many folders are stored remotely, and we can't access them directly (by using O_DIRECT | O_SYNC flags). One folder that we know for sure that is stored locally is */tmp*. This directory contains mostly files that are required temporarily and is

used by many programs to create lock files and for temporary storage of data. Therefore, our CacheFS library supports only files under /tmp. This is done by checking in the *CacheFS_open* function that the requested file is under */tmp*. If it is not, you need to return an error. See more details in the description of the *CacheFS_open* in the header file.

- **O_DIRECT | O_SYNC flags**. I highly recommend you to create a basic program that opens a file using the required flags and read from the file. After this basic experiment you will understand the restrictions better. This will separate the complications of using these flags and the cache algorithms.

- **Code Design**. As usual, make your code readable, as simple as possible, and you don't have any memory leaks. In this exercise especially I would suggest to invest in a good design. Separate between the logic of the cache algorithms and the API's implementation (e.g. by implementing them in separate files).

- **Data structure and efficiency.** Our goal is that you will implement the correct behavior of the cache algorithms. This means that you must evict the correct blocks. To do it efficiently, choose an appropriate data structure. That has been said, don't go crazy on optimizing the number of iterations over the cached items. It's not the goal of the exercise.

- Use the supplied version of *CacheFS.h*. Don't edit it, you won't submit it!

Theoretical Part (15 points)

Don't write more than a few lines per question.
Pay attention that the answers should be submitted in a separated file (*Answers.pdf)*, not in the README.

Part 1

1. Assuming we have the following processes:

Process	Arrival time	running Time	Priority (higher number- higher priority)
P1	0	10	1
P2	2	3	3
P3	4	2	2
P4	4	7	3
P5	7	1	1

   You are required to calculate the Gantt chart, turnaround time and the average wait time for the following schedulers:
   1. Round Robin (RR) with quantum=2
   2. First Come First Serve (FCFS)
   3. Shortest Remaining Time First (SRTF)
   4. Priority Scheduling
   5. Priority Scheduling with preemption

2. 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?

3. 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-cache 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?

4. 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.

5. In FBR, accesses to blocks in the "new section" doesn't increase the block's counter. Why? Which possible problem of LFU it tries to solve?

Part 2 – questions from exams

1. כמה גישות לדיסק נדרשות כדי לשמור שינוי שעושים בבית (Byte) ה- 40,000 בקובץ "os/readme.txt/"?
הניחו כי מערכת ההפעלה היא Unix, גודל בלוק (block) בדיסק הקשיח הוא בגודל 2048 בתים (2KB), גודל כל מצביע לבלוק הוא B4,
ומבנה ה-inode מכיל:
- עשרה מצביעים ישירים (direct pointers)
- מצביע אחד מסוג single indirect
- מצביע אחד מסוג double indirect
- מצביע אחד מסוג triple indirect

כמו כן הניחו כי מספר הרשומות בכל תיקייה (directory) קטן מאוד (כל מדריך של תיקייה מאוחסן בבלוק יחיד), שהקובץ קיים ושה- buffer ריק. נמקו את תשובתכם.

2. נניח כי במערכת מבוססת Unix תהליך מסוים מבצע את הפקודה seek(0) על קובץ מסוים ולאחר מכן מבצע פקודת read של בית אחד בלבד.
פרטו אילו פסיקות (interrupts) יתרחשו במערכת.
הניחו כי הקובץ נפתח בהצלחה תוך שימוש בדגלים O_DIRECT ו- O_SYNC לפני ביצוע פקודות אלו (אל תתחשבו בפתיחת הקובץ כחלק מהתשובה).

3. הנח כי ישנה מערכת עם שני סוגי עבודות: א' ו-ב'. זמן הריצה של עבודות מסוג א' הוא 1, וזמן הריצה של עבודות מסוג ב' הוא 10. כמו כן נתון כי המתזמן (scheduler) אינו יודע את זמן הריצה של עבודה נתונה (אולם הוא יודע כי העבודה יכולה להיות שייכת לסוג א' או ב' בלבד) ואינו יודע את מספר העבודות. כמו כן נתון כי המערכת תומכת בהפקעה (preemption) של עבודות.
   a. הניחו כי מספר העבודות בגודל 10 שווה למספר העבודות בגודל 1. הצע אלגוריתם הממזער את זמן ההמתנה הממוצע במערכת (average waiting time). נמק את תשובתך (אין צורך להוכיח).
   b. האם לאלגוריתם שהצעת בסעיף 1 התקורה (overhead) המינימלית האפשרית? אם כן, נמק. אם לא, הצע אלגוריתם המשיג את התקורה המינימלית (אין צורך להתחשב במדדים אחרים בסעיף זה).

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 a brief description of your code.
- ***Answers.pdf***. This pdf should contain answers (including figures) to all the questions in the theoretical part.
- ***CacheFS.cpp***, containing your implementation of the file system, and all other relevant files you implemented. Don't submit *CacheFS.h*. We use our header file.
- Your **Makefile**. Running make with no arguments should generate the ***CacheFS.a*** library. You can use the example Makefile that we provided for ex1 as a template.

Late submission policy						
Submission time	08.6, 23:55	09.6, 11:55	11.6, 23:55	12.6, 23:55	13.6, 23:55	14.6, 12:00
Penalty	0	-3	-10	-25	-40	**Course failure**