

LAB #4 – mdadm Linear Device (Caching)
CMPSC311 - Introduction to Systems Programming
Spring 2025 - Prof. Suman Saha
Due date: April 4, 2025 (11:59 PM) EST

Like all lab assignments in this class, you are prohibited from copying any content from the Internet including (discord or other group messaging apps) or discussing, sharing ideas, code, configuration, text, or anything else or getting help from anyone in or outside of the class. Consulting online sources is acceptable, but under no circumstances should anything be copied. Failure to abide by this requirement will result in penalty as described in our course syllabus.

Please note that all the subsequent labs are built on top of this lab, if you do not complete this lab, you won't be able to complete subsequent labs.

You just completed implementing mdadm and it is working. The software engineers who plan to build secure crypto wallet on top of your storage system have been torturing your storage system by throwing at it all sorts of I/O patterns, and they have been unable to find any inconsistency in your implementation. This is great, because now you have a working system, even though it may not be performant. As professor John Ousterhout of Stanford says, “the best performance improvement is the transition from nonworking state to working state”.

The software engineers are happy that your storage system is working correctly, but now they want you to make it fast as well. To this end, you are going to implement a block cache in mdadm. Caching is one of the oldest tricks in the book for reducing request latency by saving often used data in a faster (and smaller) storage medium than your main storage medium. Since we covered caching extensively in the class, we are skipping its details in this document. You must watch the lecture to understand what caching is, and how the least-recently used (LRU) algorithm that you are going to implement in this assignment works.

Overview

In general, caches store *key* and *value* pairs in a fast storage medium. For example, in a CPU cache, the key is the memory address, and the value is the data that lives at that address. When the CPU wants to access data at some memory address, it first checks to see if that address appears as a key in the cache; if it does, the CPU reads the corresponding data from the cache directly, without going to memory because reading data from memory is slow.

In a browser cache, the key is the URL of an image, and the value is the image file. When you visit a web site, the browser fetches the HTML file from the web server, parses the HTML file and finds the URLs for the images appearing on the web page. Before making another trip to retrieve the images from the web server, it first checks its cache to see if the URL appears as a key in the cache, and if it does, the browser reads the image from local disk, which is much faster than reading it over the network from a web server.

In this assignment you will implement a block cache for mdadm. In the case of mdadm, the key will be the tuple consisting of disk number and block number that identifies a specific block in JBOD, and the value will be the contents of the block. When the users of mdadm system issue `mdadm_read` call, your implementation of `mdadm_read` will first look if the block corresponding to the address specified by the user is in the cache, and if it is, then the block will be copied from the cache without issuing a slow `JBOD_READ_BLOCK` call to JBOD. If the block is not in the cache, then you will read it from JBOD and insert it to the cache, so that if a user asks for the block again, you can serve it faster from the cache.

Cache Implementation

Typically, a cache is an integral part of a storage system and it is not accessible to the users of the storage system. However, to make the testing easy, in this assignment we are going to implement cache as a separate module, and then integrate it to `mdadm_read` and `mdadm_write` calls.

Please take a look at `cache.h` file. Each entry in your cache is the following struct.

```
typedef struct {
    bool valid;
    int disk_num;
    int block_num;
    uint8_t block[JBOD_BLOCK_SIZE];
    int access_time;
} cache_entry_t;
```

The `valid` field indicates whether the cache entry is valid. The `disk_num` and `block_num` fields identify the block that this cache entry is holding and the `block` field holds the data for the corresponding block. The `access_time` field stores when the cache element was last accessed—either written or read.

The file `cache.c` contains the following predefined variables.

```
static cache_entry_t *cache = NULL;
static int cache_size = 0;
static int clock = 0;
static int num_queries = 0;
static int num_hits = 0;
```

Now let's go over the functions declared in `cache.h` that you will implement and describe how the above variables relate to these functions. You must look at `cache.h` for more information about each function.

1. `int cache_create(int num_entries);` Dynamically allocate space for `num_entries` cache entries and should store the address of the created cache in the `cache` global variable. The `num_entries` argument can be 2 at minimum and 4096 at maximum. It should also set `cache_size` to `num_entries`, since that describes the size of the cache and will also be used by other functions. `cache_size` is fixed once the cache is created. You can view it as the maximum capacity of the cache. As such, for simplicity you'd implement it as an array of size `cache_size` instead of a linked list, although the latter allows one to dynamically adding or deleting cache entries. Calling this function twice without an intervening `cache_destroy` call (see below) should fail.
2. `int cache_destroy(void);` Free the dynamically allocated space for `cache`, and should set `cache` to `NULL`, and `cache_size` to zero. Calling this function twice without an intervening `cache_create` call should fail.
3. `int cache_lookup(int disk_num, int block_num, uint8_t *buf);` Lookup the block identified by `disk_num` and `block_num` in the cache. If found, copy the block into `buf`, which cannot be `NULL`. This function must increment `num_queries` global variable every time it performs a lookup. If the lookup is successful, this function should also increment `num_hits` global variable; it should also increment `clock` variable and assign it to the `access_time` field of the corresponding entry, to indicate that the entry was used recently. We are going to use `num_queries` and `num_hits` variables to compute your cache's hit ratio.

4. `int cache_insert(int disk_num, int block_num, uint8_t *buf);` Insert the block identified by `disk_num` and `block_num` into the cache and copy `buf`—which cannot be `NULL`—to the corresponding cache entry. Insertion should never fail: if the cache is full, then an entry should be overwritten according to the LRU policy using data from this insert operation. This function should also increment and assign `clock` variable to the `access_time` of the newly inserted entry.
5. `void cache_update(int disk_num, int block_num, const uint8_t *buf);` If the entry identified by `disk_num` and `block_num` exists in cache, updates its block content with the new data in `buf`. Should also update the `access_time` if successful. This function should be called when you perform “write”.
6. `bool cache_enabled(void);` Returns true if cache is enabled (`cache_size` is larger than the minimum 2). This will be useful when integrating the cache to your `mdadm_read` and `mdadm_write` functions. That is, in your `mdadm` functions, you should call this function first whenever cache is possibly involved.

Strategy for Implementation

The tester now includes new tests for your cache implementation. You should first aim to implement functions in `cache.c` and pass all the tester unit tests. Once you pass the tests, you should incorporate your cache into your `mdadm_read` and `mdadm_write` functions—you need to implement caching in `mdadm_write` as well, because **a)** We still read in `mdadm_write` and **b)** We are going to use write-through caching policy, as described in the class. You should understand the importance and use of the cache functions. You should think about under what conditions should `cache_lookup`, `cache_insert`, and `cache_update` will be used in `mdadm.c`. Once you have incorporated cache functions in `mdadm.c`, run tester again to make sure that you still pass all the tests.

Next, try your implementation on the trace files and see if it improves the performance. To evaluate the performance, we have introduced a new cost, which is a metric into JBOD for measuring the effectiveness of your cache and is calculated based on the number of operations executed. Each JBOD operation has a different cost, and by effective caching, you reduce the number of read operations, thereby reducing your cost. Now, the tester also takes a cache size when used with a workload file, and prints the cost and hit rate at the end. The cost is computed internally by JBOD, whereas the hit rate is printed by `cache_print_hit_rate` function in `cache.c`. The value it prints is based on `num_queries` and `num_hits` variables that you should increment.

Here’s how the results look like with the reference implementation. Your implementation may produce different cost and hit rate values, depending on how you implement it (optimized or not). You are not required to output the same values, but they should be at the same magnitude as what are given. First, we run the tester on random input file:

```
$ ./tester -w traces/random-input >x
Cost: 18948700
Hit rate: nan%
```

The is 18948700, and the hit rate is undefined because we have not enabled cache. Next, we rerun the tester and specify a cache size of 1024 entries, using `-s` option:

```
$ ./tester -w traces/random-input -s 1024 >x
Cost: 17669400
Hit rate: 24.5%
```

As you can see, the cache is working, given that we have non-zero hit rate, and as a result, the cost is now reduced. Let’s try it one more time with the maximum cache size:

```
$ ./tester -w traces/random-input -s 4096 >x
Cost: 13091800
Hit rate: 87.9%
$ diff x traces/random-expected-output
$
```

Once again, we significantly reduced the cost using a larger cache. When run with trace files and cache-enabled with various cache sizes your program should show that the cost decreases with increasing cache size and the hit rate increases with increasing cache size. We also make sure that introducing caching does not violate correctness by comparing the outputs. **If introducing a cache violates correctness of your mdadm implementation, you will get a zero grade for the corresponding trace file.** Some indicators of incorrect implementation can include (but are not limited to) negative hit rate or cost, 0% hit rate with cache enabled, 100% hit rate with cache enabled, cost not decreasing, or hit rate not increasing with an increase in cache size. **If you do not incorporate cache functions correctly in mdadm.c (for both read and write) you will get 0 for trace files**

Debugging tip: Ensure that you move to the right disk and the right block not only in your function but also in jbod. Also note that you can have a 10/10 in tester and still not be implementing cache_insert properly thus failing traces.

Your workflow will consist of:

1. Editing the name of jbod object file if you are using an arm based machine like newer Macbook running on M1/M2/M3 chips rename jbod_arm64.o to jbod.o.
2. Implementing functions by modifying `cache.c` and `mdadm.c`.
3. Using `make clean`
4. Using `make` to build the `tester`
5. Ensuring there were no errors or warnings
6. Running `./tester` to see if you pass the unit tests
7. Committing and pushing your code to github
8. Running `./tester` with trace files, ensuring no output for `diff` and ensuring that cost decreases with increasing cache size.
9. Repeat steps 2-8 until you pass all the tests and the 2 trace files while reducing the cost.
10. Submit the final commit id on canvas before the due date.

Deliverables: Push your code to GitHub. Submit the commit ID of your latest code to Canvas for grading. Ensure that there is no additional text, words, or comments around the commit ID. Check canvas assignment page for more details.

Grading

Grading rubric The grading would be done according to the following rubric:

- Passing cache test cases (Totally 19 cases, but since most of them are read/write test cases for lab2 and lab3, we only count 2 write cases and 6 cache cases. The perfect score is 10 out of 10) : 70%

- Passing random and linear trace files with cache to reduce the cost: 25%
- Adding meaningful descriptive comments: 5%
- If you have **any make errors or warnings or program is stuck in a loop** then you will **receive a straight 0**. Make sure your code does not have any make error before submitting
- **If you do not submit your commit ID** on canvas you will receive a **straight 0**
- If introducing a cache violates correctness of your mdadm implementation, you will get a zero grade for the corresponding trace file.
- We would use the timestamp on Canvas when you submit your commit ID for assessing any late penalty. If you pushed your work on time on github.com but submitted the commit ID late on canvas, your work will be considered late and will incur late penalty.

Lab 4 Submission Checklist

Before submitting your Lab 4, please ensure the following:

- **Successful Compilation:** Verify that your code compiles without any errors or warnings using the make command.
- **Tester Execution:** Execute ./tester and ensure it runs without any issues, such as infinite loops, segmentation faults, or other errors.
- **Test Case Pass:** Confirm that ./tester passes all the required test cases with a perfect score of 10/10, and none of the previous test cases (read/write) fail.
- **Trace Files Validation:** Run ./tester with both linear-input and random-input trace files and ensure that the diff command yields no output, indicating successful execution.
- **Cache Performance Analysis:** Execute ./tester with trace files and cache enabled, using various cache sizes. Verify that the cost decreases with increasing cache size and that the hit rate increases with increasing cache size.
- **Comments and Documentation:** Ensure your code includes an adequate number of comments that explain your implementation choices and design rationale. This helps both the grader and yourself understand the code better and facilitates future modifications.
- **GitHub Push:** Push your code to GitHub, ensuring that you've not only committed changes but also pushed them to the repository.
- **Commit ID Submission:** Submit the commit ID (only the commit ID, without any additional text) on Canvas for tracking purposes.

Note: Incorrect implementations may exhibit negative hit rates or costs, 0% hit rate with cache enabled, 100% hit rate with cache enabled, no decrease in cost, or no increase in hit rate with an increase in cache size.