CMPSC 311 Fall 2024 Lab 4 – mdadm Linear Device (cache, cache-read, cache-write, and testing)

General Instructions

For this lab assignment, you must **not** copy any content from the Internet or discuss your work with anyone. This includes ideas, code, configuration, or text. You are also prohibited from receiving help from anyone inside or outside the class.

Tasks

For this assignment, you will be implementing a cache into your mdadm system. This block cache will store blocks as you read and write from disks. They will be stored using the *disk* and *block* that the real block is located as a key, and your read and write functions will use the cache when possible to increase read/write efficiency.

This will be done in two steps. First, you will modify cache.c in order to create your cache system. Then, you will update your read and write functions in order to utilize your new cache and test how much it increases your efficiency.

Cache Implementation

To make this assignment possible, rather than cache requiring privileged access, you will have access to the cache system directly in the form of the new cache.c and cache.h files.

Looking at the cache.h file, you can see the form of each entry in the

cache: `typedef struct{`

```
    bool valid; int disk_num;
```

```
    int block_num;
    uint8_t block[JBOD_BLOCK_SIZE];
    int clock_accesses;
} cache_entry_t;
```

Each block you store will be in the form of one of these structs as a cache_entry. 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 `clock_accesses` field stores the clock tick at which the cache block was accessed—either written or read. It is incremented every time a cache block is accessed.

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 in the `cache` global variable. 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. Calling this function twice without an intervening `cache_destroy` call (see below) should fail. The `num_entries` argument can be 2 at minimum and 4096 at maximum.

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 the `num_queries` global variable every time it performs a lookup. If

the lookup is successful, this function should also increment the `num_hits` global variable; it should also update the `clock_accesses` field of the corresponding entry to indicate that the entry was accessed recently. We are going to use the `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 **Most Recently Used** policy using data from this insert operation.

5. `void cache_update(int disk_num, int block_num, const uint8_t *buf);` If the entry exists in cache, update its block content with the new data in `buf`. Should also update the `clock_accesses`.

6. `bool cache_enabled(void);` returns true if cache is enabled. This will be useful when integrating the cache into your `mdadm_read` and `mdadm_write` functions.

7. `int cache_resize(int num_entries);` resize the cache dynamically to allocate space for `num_entries` cache entries and should store the address in the `cache` global variable just like `cache_create();`

Once you have finished the implementation of these functions, you can use the tester file to verify your implementation. Once "./tester" can be run with no failed tests, you are ready to move on to updating your read and write functions and using traces to test your cache.

Read and Write Implementation

For this step, you do not need to implement any new functions, but you are free to create helper functions as desired. The tester file will handle cache initialization, so syu will not need to use the *cache_create* or *cache_resize* in these functions. In order to get the correct results, your functions should always prioritize using the cache if it is available, and your write function should write-through, such that after the cache is updated the

disk is also updated with the new data.

Testing with Traces

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 metric into JBOD for measuring the effectiveness of your cache, which is calculated based on the number of operations executed. Each JBOD operation has a different cost, and by effectively 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 the `cache_print_hit_rate` function in cache.c. The value it prints is based on the `num_queries` and `num_hits` variables that you should increment.

Here's how the results look with the reference implementation. First, we run the tester on a random input file:

```
$ ./tester -w traces/random-input
>x Cost: 40408900 num_hits: 0,
num_queries: 0 Hit rate: -nan%
```

The cost is 40408900, 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 the `-s` option:

```
$ ./tester -w traces/random-input -s 1024 >x

Cost: 37060800 num_hits: 11679,
num_queries: 49081 Hit rate: 23.8%
```

As you can see, the cache is working, given that we have a 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: 27625200 num_hits: 44985,
num_queries: 49081 Hit rate: 91.7%

$ diff -u x traces/random-expected-output
```

```
$
```

Once again, we significantly reduced the cost by using a larger cache. This implementation of cache in `mdadm_read` is compulsory. We also make sure that introducing caching does not violate correctness by comparing the outputs. **Introducing a cache shouldn't violate the correctness of your mdadm implementation. Also, the cost should reduce, and the hit rate should increase as the cache size increases. Otherwise, you will get a zero grade for the corresponding trace file.**

Grading

Nine points of your grade will come from passing the unit tests in the tester file. You will get a point for each of the random, simple, and linear trace files if you demonstrate that your cache has reduced the cost. The assignment total is 12 and will be graded down to 10.

Grading Rubric

The grading will be done according to the following rubric:

- **Passing test cases**: 70%
- **Passing trace files**: 30%

Compilation errors, warnings, and Make errors will result in 0 points being awarded.

Penalties: 10% per day for late submission (up to 3 days). The lab assignment will not be graded if it is more than 3 days late.

Honors questions:

1. Clock Replacement Algorithm

In this question, you will implement the Clock Replacement Algorithm for overwriting blocks when inserting new blocks. Here's how it works:

1. Cached blocks are arranged in a circular list with a "clock hand" pointer that cycles through them.
2. Each block has an associated "use bit" (also called the "reference bit") that is initially set to 1 when the block is loaded or accessed.
3. When a block replacement is needed, the clock hand moves to the next block:
    a. If the block's use bit is 1, the algorithm sets it to 0 and skips to the next block.
    b. If the use bit is 0, the block is replaced, and the new block's use bit is set to 1.

Compare the hit rate of the Clock Replacement with the MRU policy, and briefly analyze the reason (Max 200 words). Please include an honor.pdf with your analysis when submitting the assignment.

2. Multi-level caching

In this project, the cache only has one level, so the information is either stored on that level or the disk. In reality, most implementations of caches have multiple levels. In this case, the higher levels will have lower read/write time cost, but will be smaller, and the lower levels are larger but more time consuming to read/write to. Additionally, it is common that when a new entry is stored, an instance is stored of it in all levels.

For this project, you should implement a 3-level cache system. From the input cache size values, you must create three caches, with the base cache being 60% of that size, second level is 30%, and top level is 10%, rounding down in case of non-integer results. As stated above, when the cache receives a new block, it should be placed on every level initially. You should also modify cache such that num_hits is a size 3 array that returns [base_hits,second_level_hits,top_level_hits].