Assignment #4 - `mdadm` Linear Device (Caching)

CMPSC311 - Introduction to Systems Programming

Fall 2023, Dr. Syed Hussain

Due date: November 13, 2023 (11:59pm EST)

You just completed implementing `mdadm`, and it is working. The software engineers who plan to build a secure crypto wallet on top of your storage system have been torturing it 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 the nonworking state to the 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 **most-recently used** (MRU) algorithm that you are going to implement in this assignment works. Specifically, you will be implementing a look-aside cache with a write-through write policy.

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 the 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 the `mdadm` system issue a `mdadm_read` call, your implementation of `mdadm_read` will first look to see 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 into 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 into `mdadm_read` and `mdadm_write` calls.

Please take a look at the cache.h file. Each entry in your cache has the following structure:

```
typedef struct{

      bool valid;
      int disk_num;
      int block_num;
      uint8_t block[JBOD_BLOCK_SIZE];
      int clock_accesses;
} 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 `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();`

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 test 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 we are going to use a write-through caching policy, as described in the class. Once you do that, 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 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.

Extra Credit

1 point of extra credit for incorporating cache into `mdadm_write` and not violating the correctness of the mdadm implementation.