CMPSC 311 Fall 2024 Lab 3 – mdadm Linear Device (writes, permissions 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

As part of this assignment, you are required to implement **3 functions**. Since you performed a read operation in the previous assignment, you will now be implementing a write operation in a block. Additionally, you will implement two methods to manage the write permissions of the disk. The description of each method is given below:

1. `mdadm_write`

Function prototype:

```
int mdadm_write(uint32_t start_addr, uint32_t write_len, const
uint8_t write_buf);
```

The function prototype is very similar to that of the `mdadm_read` function, which you have already implemented. The `mdadm_write` method should write `write_len` bytes from the user-supplied `write_buf` buffer to your storage system, starting at address `start_addr`.

You may notice that the `write_buf` parameter now has a `const` specifier. This indicates that it is an input parameter; that is, `mdadm_write` should only read from this parameter and not modify it. It is a good practice to specify the `const` specifier for your input parameters that are arrays or structs. On success, return the amount of bytes written. There are a few conditions that should not allow `mdadm_write` to execute:

- Similar to `mdadm_read`, writing to an out-of-bound linear address should fail. Return -1 in this case.
- Writes larger than 1024 bytes should also fail; in other words, `write_len` can be at most **1024**. The function should return -2 in this case.
- Writing when the system is unmounted should fail. Return -3 in this case.
- There may be additional restrictions that should cause `mdadm_write` to fail. You'll encounter these cases as you pass the tests. Return -4 for them.

Once you implement this function, you will have the basic functionality of your storage system in place. We have expanded the tester to include new tests for the write operations, in addition to existing read operations. You should try to pass these write tests first.

---

2. `mdadm_write_permission`

Function Prototype:

```
int mdadm_write_permission(void);
```

This method has no function arguments. The result of calling this function is to allow write operations on the storage system once the function is called. The `mdadm_write` method should return -5 if the user does not have the write permissions.

---

3. `mdadm_revoke_write_permission`

Function Prototype:

```
int mdadm_revoke_write_permission(void);
```

Similar to the previous method, calling this method should block all write operations on the storage device. The storage device would become a read-only device once this is called.

As you might expect, you must turn write permission **ON** before writing to the system. You should also always check if this permission is on each time you write to the system.

Interaction with the Storage System

Just as all other functions that you have been required to implement, you will need to use `jbod_operation(op, *block)` to interact with the storage system. The following **ENUM** commands are provided to you:

- **`JBOD_WRITE_BLOCK`:** Writes the data in the block buffer into the block in the current I/O position. The buffer pointed to by `block` must be of block size, that is **256 bytes**. More importantly, after this operation completes, the `CurrentBlockID` in I/O position is incremented by **1**; that is, the next I/O operation will happen on the next block of the current disk. When the command field of `op` is set to this command, all other fields in `op` are ignored by the JBOD driver.
- **`JBOD_WRITE_PERMISSION`:** Sets the write permission to **1** so that writing will be allowed. When the command field of `op` is set to this, all other fields in `op` are ignored by the JBOD driver. The block argument is passed as **NULL**. Return **0** on success and **-1** on failure.
- **`JBOD_REVOKE_WRITE_PERMISSION`:** Sets the write permission to **-1** so that writing will no longer be allowed. When the command field of `op` is set to this, all other fields in `op` are ignored by the JBOD driver. The block argument is passed as **NULL**. Return **0** on success and **-1** on failure.

HINT: Check your write permission in your code before performing any write operation.

Testing Using Trace Replay

As we discussed before, your **mdadm** implementation is a layer right above **JBOD**, and the purpose of **mdadm** is to unify multiple small disks under a unified storage system with a single address space. An application built on top of **mdadm** will issue a **mdadm_mount**, **mdadm_write_permission**, and then a series of **mdadm_write** and **mdadm_read** commands to implement the required functionality, and eventually, it will issue the **mdadm_unmount** command. Those read/write commands can be issued at

arbitrary addresses with arbitrary payloads, and our small number of tests may have missed corner cases that may arise in practice.

Therefore, in addition to the unit tests, we have introduced trace files, which contain the list of commands that a system built on top of your **mdadm** implementation can issue. We have also added to the tester functionality to replay the trace files. Now, the tester has two modes of operation.

If you run it without any arguments, it will run the unit tests:

```
$ ./tester
```

If you run it with `-w pathname` arguments, it expects the pathname to point to a trace file that contains the list of commands. In your repository, there are three trace files under the **traces** directory: **simple-input**, **linear-input**, and **random-input**. Let's look at the contents of one of them using the `head` command, which shows the first 10 lines of its argument:

```
$ head traces/simple-input

MOUNT WRITE_PERMIT WRITE 0 256 0

READ 1006848 256 0

WRITE 1006848 256 93

WRITE 1007104 256 94

WRITE 1007360 256 95

READ 559872 256 0

WRITE 559872 256 139

READ 827904 256 0

WRITE 827904 256 162
```

The first command mounts the storage system. The second command is a write command, and the arguments are similar to the actual **mdadm_write** function arguments; that is, write at address 0, 256 bytes of 0. The third command reads 256 bytes from address 1006848 (the third argument to **READ** is ignored). And so on. You can replay them on your implementation using the tester as follows:

```
$ ./tester -w traces/simple-input

SIG(disk,block) 0 0 : 0xb3 0x76 0x88 0x5a 0xc8 0x45 0x2b 0x6c
0xbf 0x9c

SIG(disk,block) 0 1 : 0xb3 0x76 0x88 0x5a 0xc8 0x45 0x2b 0x6c
0xbf 0x9c

SIG(disk,block) 0 2 : 0xb3 0x76 0x88 0x5a 0xc8 0x45 0x2b 0x6c
0xbf 0x9c

SIG(disk,block) 0 3 : 0xb3 0x76 0x88 0x5a 0xc8 0x45 0x2b 0x6c
0xbf 0x9c

...
```

If one of the commands fails, for example, because the address is out of bounds, then the tester aborts with an error message indicating on which line the error occurred. If the tester can successfully replay the trace until the end, it takes the cryptographic checksum of every block of every disk and prints them out on the screen, as shown above.

Now you can use this information to determine if the final state of your disks is consistent with the final state of the reference implementation, if the above trace was replayed on a reference implementation. You can do that by comparing your output to that of the reference implementation. The files that contain the corresponding cryptographic checksums from the reference implementation are also under the **traces** directory and they end with **-expected-output**. For example, here's how you can test if your implementation's trace output matches that of the reference implementation's output for the **simple-input** trace:

```
$ ./tester -w traces/simple-input >my-output
```

```
$ diff -u my-output traces/simple-expected-output
```

The first line replays the trace file and redirects the output to **my-output** file in the current directory, while the second line runs the **diff** tool, which compares the contents of the two files – when the files are identical, no output is displayed, which means your implementation's final state after the commands in the trace file matches the reference implementation's state. If there is a difference in the **diff** command output, then there is a bug in your implementation; you can see which block contents differ, which may help you with debugging.

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. Answer the following question.

   Consider the situation: you are writing a large chunk to the JBOD system, unfortunately, the JBOD system crashes when your write operation is in progress. After rebooting the JBOD system, you want to recover the JBOD system to the status before the failed write operation. However, a portion of the new chunk has been written to the disk.

Propose a new JBOD operation that can recover the JBOD system to the status before the write operation in which it crashed. Describe your method in detail and use examples to prove why it can recover the JBOD system in a correct manner. **Please include an honor.pdf with a description of your method when submitting the assignment.**

2. Coding.

   Modify the mdadm_write function to simulate the method you propose. **We won't test your code with real JBOD system crashes.** You can just print out the steps you need to support the failure recovery.