Lab Assignment #3-mdadm Linear Device (Writes, Permissions and Testing)

CMPSC311 - Introduction to Systems Programming

Fall 2023 - Prof. Syed Rafiul Hussain

Due date: October 24, 2023(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. Failure to abide by this requirement will result in penalty as described in our course syllabus.

Your internship is going great. You have gained experience with C programming, you have experienced your first segmentation faults, and you've come out on top. You are brimming with confidence and ready to handle your next challenge.

Tasks

As part of this assignment, you are required to implement 3 functions. Since you did a read operation in the previous assignment, you will now be implementing a write operation in a block. You will also be implementing 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)`\

As you can tell, the function prototype is very similar to that of the mdadm_read function, which you have already implemented. The `mdadm_write` method should `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. We put the `const` there to emphasize 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 `const` specifier for your in parameters that are arrays or structs. Similar to `mdadm_read`, writing to an out-of-bound linear address should fail. Write larger than 1024 bytes should fail; in other words, `write_len` can be 1024 at most. On success return the amount of bytes written. There are a few more restrictions that you will find out as you try to pass the tests. Once you implement the above function, you 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);` \

As you can tell 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 an error if the user does not have the write permissions.

3. `mdadm_revoke_write_permission(void)`
   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.

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 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 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 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 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 a 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, 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 saying on which line the error happened. 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 above. Now you can use this information to tell 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 reference implementation are also under traces directory and they end with - expected-output. For example, here's how you can test if your implementation's trace output matches with that of 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 to files contents – 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 blocks contents differ and that may help you with debugging.

Grading rubric

The grading would be done according to the following rubric:

- Passing test cases 70%
- Passing trace files 30%
- **Compilation error, 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.