Lab #3 – `mdadm` Linear Device (Writes and Testing)
CMPSC311 - Introduction to Systems Programming
Spring 2024 - Prof. Sencun Zhu
Due date: March 15, 2024 (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.

Note: Before proceeding to lab 3, ensure that you have completed the read functionality from lab 2. You will receive credit for completing the last two test cases for read in lab 3 as well.

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.

Implementing `mdadm_write`

Your next job is to implement write functionality for `mdadm` and then thoroughly test your implementation. Specifically, you will implement the following function:

```
int mdadm_write(uint32_t addr, uint32_t len, const uint8_t *buf)
```

As you can tell, it has an interface that is similar to that of the `mdadm_read` function, which you have already implemented. Specifically, it writes `len` bytes from the user-supplied `buf` buffer to your storage system, starting at address `addr`. You may notice that the `buf` parameter now has a `const` specifier. We put the `const` there to emphasize that it is an *in* 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. A read larger than 1,024 bytes should fail; in other words, `len` can be 1,024 at most. 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.

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` 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 introduces 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 function-

ality 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
/*****************************************************************/
Please ensure there were no warnings when you compiled your program
If you have any make errors or warnings then you will receive a straight 0.
Please ensure there are sufficent comments in your code
Do not foget to add your name and date in comments on top of your mdadm.c
/*****************************************************************/


Testing your code now:running test_mount_unmount: passed
running test_read_before_mount: passed
running test_read_invalid_parameters: passed
running test_read_within_block: passed
running test_read_across_blocks: passed
running test_read_three_blocks: passed
running test_read_across_disks: passed
running test_write_before_mount: passed
running test_write_invalid_parameters: passed
running test_write_within_block: passed
running test_write_across_blocks: passed
running test_write_three_blocks: passed
running test_write_across_disks: passed
Total score: 10/10
```

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 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 bytes with contents 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 0xed ..
SIG(disk,block) 0 1: 0xb3 0x76 0x88 0x5a 0xc8 0x45 0x2b 0x6c 0xbf 0x9c 0xed ..
SIG(disk,block) 0 2: 0xb3 0x76 0x88 0x5a 0xc8 0x45 0x2b 0x6c 0xbf 0x9c 0xed ..
SIG(disk,block) 0 3: 0xb3 0x76 0x88 0x5a 0xc8 0x45 0x2b 0x6c 0xbf 0x9c 0xed ..
...
```

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. **No output for diff is considered passed. Just running a trace file with tester is not considered pass.**
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.For example you might see something like:

```
-SIG(disk,block) 5 239 : 0x8d 0x95 0xbf 0x5c 0xae 0xec 0xe0 0x0d 0x6a 0xa6
+SIG(disk,block) 5 239 : 0x42 0xae 0xb0 0xe5 0x68 0x15 0x81 0x7e 0x20 0x15
```

This means that signature of disk 5 block 239 does not match with the reference implementation. Your code did not write the right content on disk 5, block 239. 0x8... part is the signature of the content of the block and does not tell you the exact content of the block.
Based on diff's output you know what disk and block numbers were not written correctly. This likely means there was an issue in your logic or you missed implementing a particular edge case.

Debugging tips:

- Without gdb: Based on diff's output you know what disk and block numbers were not written correctly. This likely means there was an issue in your logic or you missed implementing a particular edge case. You can find out the exact address that is causing the issue by strategically using printf statements to printout a) The addr and len you are dealing with (ideally at being of madm_write function), b) the length you are using to copy into buffer every time you use memcpy or before you write using jbod write operation c) print out disk number and block number before your write to ensure you are writing at the right disk and right block.

- With gdb: You can run the tester files in gdb. You can do it with the following commands:

```
 gdb ./tester
(gdb) set args -w traces/simple-input
(gdb) r
```

This should run the code using the tracer files as input. You can replace simple with linear and random as you need.

If you are able to pass test cases but not the trace files, use gdb to debug your code. If you come across a failure while running in gdb, try the command `where`, it should show where in your code the issue appears.

You 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 write functions by modifying `mdadm.c` from your Lab2.
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 and ensuring no output for `diff`
9. Repeat steps 2-8 until you pass all the tests and all 3 trace files.
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 rubric The grading would be done according to the following rubric:

- Passing test cases 70%
  Note that you get points for all the write tests and 2 read tests(test_read_three_blocks and test_read_across_disks)
- Passing 3 trace files 25% (simple-input 7%, linear-input 8%, and random-input 10%)
- 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**

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.