Lab #2 – `mdadm` Linear Device (Basic Functionality)

CMPSC311 - Introduction to Systems Programming
Spring 2025 - Prof. Suman Saha

Checkpoint: February 23, 2025 (11:59 PM) EST
Due date: February 28, 2025 (11:59 PM) EST

Like all lab assignments in this class, you are prohibited from copying any content from the Internet or discussing, sharing ideas, code, configuration, text, or anything else or getting help from anyone (AI or human) in or outside of the class. Failure to abide by this requirement will result in penalty as described in our course syllabus.

Checkpoint 1: **You should pass the first two tests, test_mount_unmount(), and test_read_before_mount() by February 23, 11:59 PM EST. Otherwise you will get 50% penalty even if you pass all tests by the due date.**

Please note that all the subsequent labs are built on top of this lab, if you do not complete this lab, you wont be able to complete subsequent labs.

Today is the first day of your summer internship at a cryptocurrency startup. Before you join, the marketing team decided that they want to differentiate their product by emphasizing on security. On the same day that you join the company, the shipment of 16 military-grade, nuclear bomb-proof hard disks arrives. They are supposed to replace the existing commercial-grade hard disks and will be used to store the most critical user data—cryptocurrency wallets. However, the disk company focuses on physical security and doesn't invest much in software. They provide their disks as a JBOD (Just a Bunch of Disks), which is a storage architecture consisting of numerous disks inside of a single storage enclosure. They also provide a user manual along with the shipment:

Bits	Width	Field	Description
24-31	8	BlockID	Block address within the disk
20-23	4	DiskID	This is the ID of the disk to perform operation on
6-19	14	Reserved	Unused bits (for now)
0-5	6	Command	This is the command to be executed by JBOD.

Table 1: JBOD operation format

Thank you for purchasing our military-grade, nuclear bomb-proof hard disks, built with patented NASA technologies. Each of the disks in front of you consists of 256 blocks, and each block has 256 bytes, coming to a total of $256 \times 256 = 65{,}536$ bytes per disk. Since you bought 16 disks, the combined capacity is $16 \times 65{,}536 = 1{,}048{,}576$ bytes = 1 MB. We provide you with a device driver with a single function that you can use to control the disks.

```
int jbod_operation(uint32_t op, uint8_t *buff);
```

This function returns 0 on success and -1 on failure. It accepts an operation through the `op` parameter, the format of which is described in Table 1, and a pointer to a buffer. **The command field can be one of the following commands**, which are declared as a C `enum` type in the header that we have provide to you:

1. `JBOD_MOUNT`: mount all disks in the JBOD and make them ready to serve commands. This is the first command that should be called on the JBOD before issuing any other commands; all commands before it will fail. When the command field of `op` is set to this command, all other fields in `op` are ignored by the JBOD driver. Similarly, the `buff` argument passed to `jbod_operation` can be NULL.

2. `JBOD_UNMOUNT`: unmount all disks in the JBOD. This is the last command that should be called on the JBOD; all commands after it will fail. When the command field of `op` is set to this command, all other fields in `op` are ignored by the JBOD driver. Similarly, the `buff` argument passed to `jbod_operation` can be NULL.

3. `JBOD_SEEK_TO_DISK`: seeks to a specific disk. JBOD internally maintains an *I/O position*, a tuple consisting of {**CurrentDiskID**, **CurrentBlockID**}, which determines where the next I/O operation will happen. This command seeks to the beginning of disk specified by DiskID field in `op`. In other words, it modifies I/O position: it sets CurrentDiskID to DiskID specified in `op` and it sets CurrentBlockID to 0. When the command field of `op` is set to this command, the BlockID field in `op` is ignored by the JBOD driver. Similarly, the `buff` argument passed to `jbod_operation` can be NULL.

4. `JBOD_SEEK_TO_BLOCK`: seeks to a specific block in current disk. This command sets the CurrentBlockID in *I/O position* to the block specified in BlockID field in `op`. When the command field of `op` is set to this command, the DiskID field in `op` is ignored by the JBOD driver. Similarly, the `buff` argument passed to `jbod_operation` can be NULL.

5. `JBOD_READ_BLOCK`: reads the block in current I/O position into the buffer specified by the `block` argument to `jbod_operation`. The buffer pointed by `buff` must be of block size, that is 256 bytes. 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* unless you specify a new DiskID or BlockID. When the command field of `op` is set to this command, all other fields in `op` are ignored by the JBOD driver.

6. `JBOD_WRITE_BLOCK`: writes the data in the `buff` buffer into the block in the current I/O position. The buffer pointed by `buff` must be of block size, that is 256 bytes. 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 unless you specify a new DiskID or BlockID. When the command field of `op` is set to this command, all other fields in `op` are ignored by the JBOD driver.

After you finished your onboarding session with HR and enjoyed the free lunch with your new colleagues, you received the following email from the manager of the team.

Welcome, to the team! Here's your task for the next two weeks. You will be working on integrating JBOD into our existing storage system. Specifically, you will implement one of the functionalities of the `mdadm` utility in Linux. `Mdadm` stands for multiple disk and device administration, and it is a tool for doing cool tricks with multiple disks. You will implement one of such tricks supported by `mdadm`, called *linear device*. A linear device makes multiple disks appear as a one large disk to the operating system. In our case, we will use your program to configure 16 disks of size 64 KB as a single 1 MB disk.

Your task is to implement an mdadm linear device, which will merge these 16 disks into a single logical disk with a linear address space. This linear address space allows for reading and writing bytes within the range of 0 to 1,048,575. To clarify, the mdadm linear device should map this linear address space to the disks sequentially. For instance, addresses 0 to 65,535 correspond to disk 0, addresses 65,536 to 131,071 correspond to disk 1, and so forth.

Note: Before implementing the functions, fill out correct values for JBOD_NUM_DISKS, JBOD_-DISK_SIZE, JBOD_BLOCK_SIZE, JBOD_NUM_BLOCKS_PER_DISK in jbod.h file. It is required for implementing below functions.

Below are the functions you need to implement.

`int mdadm_mount(void)`: Mount the linear device; now `mdadm` user can run read and operations on the linear address space that combines all disks. It should return 1 on success and -1 on failure. Calling this function the second time without calling `mdadm_unmount` in between, should fail.

`int mdadm_unmount(void)`: Unmount the linear device; now all commands to the linear device should fail. It should return 1 on success and -1 on failure. Calling this function the second time without calling `mdadm_mount` in between, should fail.

`int mdadm_read(uint32_t addr, uint32_t len, uint8_t *buf)`: Read `len` bytes into `buf` starting at `addr`. It returns -1 on failure and actual length of read data in case of success. Read from 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.
Tip: It is import to make sure you are at the correct disk and block before calling the read function.
Good luck with your task!

Now you are all pumped up and ready to make an impact in the new company. You spend the afternoon with your mentor, who goes through the directory structure and the development procedure with you:

1. `jbod.h`: The interface of JBOD. You will use the constants defined here in your implementation.
2. `jbod.o`: The object file containing the JBOD driver.
3. `mdadm.h`: A header file that lists the functions you should implement.
4. `mdadm.c`: Your implementation of mdadm functions.
5. `tester.h`: Tester header file.

6. `tester.c`: Unit tests for the functions that you will implement. This file will compile into an executable, `tester`, which you will run to see if you pass the unit tests.

7. `util.h`: Utility functions used by JBOD implementation and the tester.

8. `util.c`: Implementation of utility functions.

9. `Makefile`: instructions for compiling and building `tester` used by the `make` utility.

You workflow will consist of:

1. Editing the values in jbod.h

2. Editing the name of the jbod object file. If you are using an arm-based machine like a newer Macbook running on M1/M2/M3 chips, rename jbod_arm64.o to jbod.o otherwise, rename jbod_x86.o to jbod.o

3. Implementing functions by modifying `mdadm.c`. If it is your first time editing mdadm.c make sure to type your name in the space provided in mdadm.c

4. Using `make clean`

5. Using `make` to build the `tester`

6. Ensuring there were no errors or warnings

7. Running `./tester` to see if you pass the unit tests

8. **Committing and pushing your code to github**

9. Repeat steps 3-8 until you pass all the tests.

10. Submit the final commit id on canvas before the due date.

Although you only need to edit `mdadm.c` for successfully completing the assignment, you can modify any file you want if it helps you in some way. When testing your submission, however, **we will use the original forms of all files except `mdadm.c` and `mdadm.h`.** Remember that you are free to create helper functions if that helps you in mdadm.c (e.g., if you want to have a helper function to determine which block and disk correspond to a specific linear address).

Debugging tip: Check the return value of `jbod_operation` to know if what you were trying to do was successful. If you are having issues with read ensure that the mount operation was successful. If you didn't mount the disks properly, all read operations will also fail

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 95%

- Adding meaningful descriptive comments 5%

- If you have **any make errors or warnings or if 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**

- We would use the timestamp on Canvas when you submit your commit ID for assessing any late penalty. If you pushed your work on time on github.com but submitted the commit ID late on canvas, your work will be considered late and will incur late penalty.

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. 50% penalty if first checkpoint is not passed by Feb 23 2025, 11:59 pm (No late submission for the checkpoint).

Asking for help: If you are stuck and need assistance, here's the recommended approach:

1. Check Campuswire

    - Start by searching Campuswire for similar issues. Often, other students may have encountered and resolved the same problem.
    - If you don't find an answer, you can create a new post describing your issue.
    Important: Do not post your code directly on Campuswire. If your question is code-specific and requires a TA to review your work, you may share your commit ID. The commit ID is unique and only visible to you and the TAs, ensuring the integrity of your work

    .

2. Email or Visit Office Hours

    - If Campuswire doesn't resolve your issue, email your assigned TA or visit their office hours.
    - Do not email multiple TAs separately or use BCC. Instead, email one TA, and if you do not hear back from them in a timely manner, you can send an email to other TAs in the same email chain and clearly indicate your issue. This ensures transparency and avoids duplication of effort among TAs.

Following these guidelines will help ensure timely and effective assistance while maintaining fairness and efficiency for all.