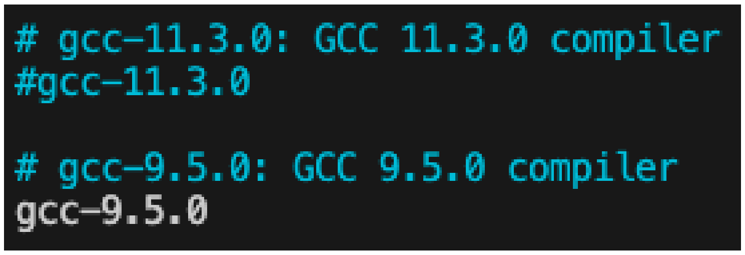
Lab 2 – `mdadm` Linear Device (Read Functionality)

CMPSC311 - Introduction to Systems Programming
Fall 2023 - Prof. Syed Rafiul Hussain
Due date: October 2, 2023 (11:59 PM) EST

Like all lab assignments in this class, you are prohibited from copying any content from the Internetor discussing, sharing ideas, code, configuration, text, or anything else or getting help from anyone in or outside of the class.

Use gcc-9 to compile. To change to gcc-9 on lab machines that you have logged in follow the below steps.

1. Run the command '*/home/software/user_conf/bin/new_soft*'(This command will create a file ~/.software)
2. Run the command *'vi ~/.software'* (This will open the created file in vi editor, you will have to edit this file to change gcc version)
3. Find the lines for gcc version in the file you have opened with vi editor and uncomment the gcc-9.5.0. It will look like this

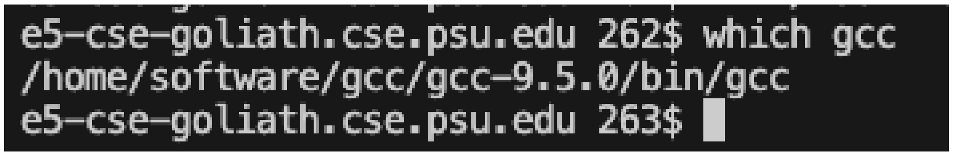


```
# gcc-11.3.0: GCC 11.3.0 compiler
#gcc-11.3.0

# gcc-9.5.0: GCC 9.5.0 compiler
gcc-9.5.0
```

NOTE : Make sure only 'gcc-9.5.0' is uncommented and all the above 3 lines are still commented with #)

4. Logout of the machine by closing VSCode
5. Reopen the VSCode and connect to lab machine
6. Run command '*which gcc*'
7. You should be able to see



```
e5-cse-goliath.cse.psu.edu 262$ which gcc
/home/software/gcc/gcc-9.5.0/bin/gcc
e5-cse-goliath.cse.psu.edu 263$ 
```

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 thesame 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**
0-3	4	DiskID	This is the ID of the disk to perform operation on
4-11	8	BlockID	This is the ID of the block within disk to perform operation on
12-19	8	Command	This is the command to be executed by JBOD.
20-31	12	Reserved	Unused bits (for now)

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 *i* blocks, and each block has 256 bytes. Since you bought *j* disks, the combined capacity is *i x j x* 256= 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. *i* and *j* are unknown and integers.

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

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 `JBOD_MOUNT` will fail. When the command field of `op` is set to this command, all other fields in `op` are ignored bythe JBOD driver. Similarly, the `block` 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 `block` 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 `block` 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 Current-BlockID 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 `block` 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 `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 JBODdriver.

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 *j* disks of size *k* KB as a single 1 MB disk. Below are the functions you need to implement.

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. The right values are regarding the unknown i, j and k.

`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 start_addr, uint32_t read_len, uint8_t *read_buf)`: Read read_len bytes into read_buf starting at start_addr. Read from an out-of -bound linear address should fail. A read larger than 1024 bytes should fail; in other words, read_len can be 1,024 at most. There are a few more restrictions that you will findout as you try to pass the tests.

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. (NOTE : you have to edit the right values in this file)
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. (NOTE : You will edit this file with all your implementations)
5. `tester.h`: Tester header file.
6. `tester.c`: Unit tests for the functions that you will implement. This file will compile into an executa- ble, `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.

Your workflow will consist of

(1) Editing with right values in `jbod.h`

(2) Implementing functions by modifying `mdadm.c`

(3) Run *make clean* to remove object files

(4) Run *make* command to build the tester

(5) Run *./tester* to see if you pass the unit tests

Repeating these steps until you pass all the tests. Although you only need to edit jbod.h and 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 jbod.h, 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).

Grading rubric : The grading would be done according to the following rubric:

- If you have any make errors then you will receive a straight 0. Make sure your code does not have any make error before submitting
- We have 10 test cases for this Assignment and each test case will carry 1 points each.
- NOTE : We recommend you to add comments in your code for better readability.

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.