# File System Interface Project Descriptions

Yicheng Rong

2022/08/17

## Part 1: Basic Functionality

Today, the disk company focuses on physical security and doesn't invest much in software. If 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 |
|---|---|---|---|
| 26-31 | 6 | Command | This is the command to be executed by JBOD. |
| 22-25 | 4 | DiskID | This is the ID of the disk to perform operation on |
| 8-21 | 14 | Reserved | Unused bits (for now) |
| 0-7 | 8 | BlockID | Block address within the disk |

Each of the disks in front of you consists of 256 blocks, and each block has 256 bytes, coming to a total of 256 × 256 = 65,536 bytes per disk. Since you bought 16 disks, the combined capacity is 16 × 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 *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:

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 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 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 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 JBOD driver.

6. 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.

File Descriptions:
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.

## Part 2: Writes and Testing

**Implementing mdadm_write**

My next job is to implement write functionality for mdadm and then thoroughly test my implementation. Specifically, I implemented 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 I have already implemented. Specifically, it writes len bytes from the user-supplied buf buffer to our 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 I implemented 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, my 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 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
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
```

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 0x81
0xb1 0x08 0x0b
SIG(disk,block) 0 1 : 0xb3 0x76 0x88 0x5a 0xc8 0x45 0x2b 0x6c 0xbf 0x9c 0xed 0x81
0xb1 0x08 0x0b
SIG(disk,block) 0 2 : 0xb3 0x76 0x88 0x5a 0xc8 0x45 0x2b 0x6c 0xbf 0x9c 0xed 0x81
0xb1 0x08 0x0b
SIG(disk,block) 0 3 : 0xb3 0x76 0x88 0x5a 0xc8 0x45 0x2b 0x6c 0xbf 0x9c 0xed 0x81
0xb1 0x08 0x0b ...
```

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
```

## Part 3: Caching

**Overview**

In general, caches store key and value pairs in a fast storage medium. For example, in a CPU cache, the key is the memory address, and the value is the data that lives at that address. When the CPU wants to access data at some memory address, it first checks to see if that address appears as a key in the cache; if it does, the CPU reads the corresponding data from the cache directly, without going to memory because reading data from memory is slow.

In a browser cache, the key is the URL of an image, and the value is the image file. When you visit a web site, the browser fetches the HTML file from the web server, parses the HTML file and finds the URLs for the images appearing on the web page. Before making another trip to retrieve the images from the web server, it first checks its cache to see if the URL appears as a key in the cache, and if it does, the browser reads the image from local disk, which is much faster than reading it over the network from a web server.

In this case I implement a block cache for mdadm. In the case of mdadm, the key will be the tuple consisting of disk number and block number that identifies a specific block in JBOD, and the value will be the contents of the block. When the users of mdadm system issue mdadm_read call, the implementation of mdadm_read will first look if the block corresponding to the address specified by the user is in the cache, and if it is, then the block will be copied from the cache without issuing a slow JBOD_READ_BLOCK call to JBOD. If the block is not in the cache, then you will read it from JBOD and insert it to the cache, so that if a user asks for the block again, you can serve it faster from the cache

**Cache Implementation**

Typically, a cache is an integral part of a storage system and it is not accessible to the users of the storage system. However, to make the testing easy, in this project I'm going to implement cache as a separate module, and then integrate it to mdadm_read and mdadm_write calls.

Please take a look at cache.h file. Each entry in mycache is the following struct.

```
typedef struct {
        bool valid;
        int disk_num;
        int block_num;
        uint8_t block[JBOD_BLOCK_SIZE];
        int access_time;
        } cache_entry_t;
```

The valid field indicates whether the cache entry is valid. The disk_num and block_num fields identify the block that this cache entry is holding and the block field holds the data for the corresponding block. The access_time field stores when the cache element was last accessed—either written or read.

The file cache.c contains the following predefined variables.

```
static cache_entry_t *cache = NULL;
static int cache_size = 0;
static int clock = 0;
static int num_queries = 0;
static int num_hits = 0;
```

I created the following functions to achieve these goals.

1. int cache_create(int num_entries); Dynamically allocate space for num_entries cache entries and should store the address of the created cache in the cache global variable. The num_-entries argument can be 2 at minimum and 4096 at maximum. It should also set cache_size to num_entries, since that describes the size of the cache and will also be used by other functions. cache_size is fixed once the cache is created. You can view it as the maximum capacity of the cache. As such, for simplicity you'd implement it as an array of size cache_size instead of a linked list, although the latter allows one to dynamically adding or deleting cache entries. Calling this function twice without an intervening cache_destroy call (see below) should fail.

2. int cache_destroy(void); Free the dynamically allocated space for cache, and should set cache to NULL, and cache_size to zero. Calling this function twice without an intervening cache_- create call should fail.

3. int cache_lookup(int disk_num, int block_num, uint8_t *buf); Lookup the block identified by disk_num and block_num in the cache. If found, copy the block into buf, which cannot be NULL. This function must increment num_queries global variable every time it performs a lookup. If the lookup is successful, this function should also increment num_hits global variable; it should also increment clock variable and assign it to the access_time field of the corresponding entry, to indicate that the entry was used recently. We are going to use num_queries and num_hits variables to compute your cache's hit ratio.

4. int cache_insert(int disk_num, int block_num, uint8_t *buf); Insert the block identified by disk_num and block_num into the cache and copy buf—which cannot be NULL—to the corresponding cache entry. Insertion should never fail: if the cache is full, then an entry should be overwritten according to the LRU policy using data from this insert operation. This function should also increment and assign clock variable to the access_time of the newly inserted entry.

5. void cache_update(int disk_num, int block_num, const uint8_t *buf); If the entry exists in cache, updates its block content with the new data in buf. Should also update the access_time if successful.

6. bool cache_enabled(void); Returns true if cache is enabled (cache_size is larger than the minimum 2). This will be useful when integrating the cache to your mdadm_read and mdadm_write functions. That is, in your mdadm functions, you should call this function first whenever cache is possibly involved.

**Strategy for Implementation**

The tester now includes new tests for your cache implementation. You should first aim to implement functions in cache.c and pass all the tester unit tests. Once you pass the tests, you should incorporate your cache into your mdadm_read and mdadm_write functions—you need to implement caching in mdadm_write as well, because we are going to use write-through caching policy, as described in the class. Once you do that, make sure that you still pass all the tests.

Next, try your implementation on the trace files and see if it improves the performance. To evaluate the performance, we have introduced a new cost is a metric into JBOD for measuring the effectiveness of your cache, which is calculated based on the number of operations executed. Each JBOD operation has a different cost, and by effective caching, you reduce the number of read operations, thereby reducing your cost. Now, the tester also takes a cache size when used with a workload file, and prints the cost and hit rate at the end. The cost is computed internally by JBOD, whereas the hit rate is printed by cache_print_hit_rate function in cache.c. The value it prints is based on num_queries and num_hits variables that you should increment.

Here's how the results look like with the reference implementation. Your implementation may produce different cost and hit rate values, depending on how you implement it (optimized or not). You are not required to output the same values, but they should be at the same magnitude as what are given. First, we run the tester on random input file:

```
$ ./tester -w traces/random-input >x
Cost: 18948700
Hit rate: -nan%
```

The cost is 18948700, and the hit rate is undefined because we have not enabled cache. Next, we rerun the tester and specify a cache size of 1024 entries, using -s option:

```
$ ./tester -w traces/random-input -s 1024 >x
Cost: 17669400
Hit rate: 24.5%
```

As you can see, the cache is working, given that we have non-zero hit rate, and as a result, the cost is now reduced. Let's try it one more time with the maximum cache size:

```
$ ./tester -w traces/random-input -s 4096 >x
Cost: 13091800
Hit rate: 87.9%
$ diff x traces/random-expected-output $
```

Once again, we significantly reduced the cost using a larger cache. We also make sure that introducing caching does not violate correctness by comparing the outputs. If introducing a cache violates correctness of your mdadm implementation, you will get a zero grade for the corresponding trace file.

## Part 4: Networking

Adding a cache to our mdadm system has significantly improved its latency and reduced the load on the JBOD. In this final step of my project, I implemented a client component of this protocol that will connect to the JBOD server and execute JBOD operations over the network. As the company scales, they plan to add multiple JBOD systems to their data center. Having networking support in mdadm will allow the company to avoid downtime in case a JBOD system malfunctions, by switching to another JBOD system on the fly

Currently,my mdadm code has multiple calls to jbod_operation, which issue JBOD commands to a locally attached JBOD system. In my new implementation, I replace all calls to jbod_operation with jbod_client_operation, which will send JBOD commands over a network to a JBOD server that can be anywhere on the Internet (but will most probably be in the data center of the company). I also implemented several support functions that will take care of connecting/disconnecting to/from the JBOD server.

**Protocol**

The protocol defined by the JBOD vendor has two messages. The JBOD request message is sent from your client program to the JBOD server and contains an opcode and a buffer when needed (e.g., when your client needs to write a block of data to the server side jbod system). The JBOD response message is sent from the JBOD server to your client program and contains an opcode and a buffer when needed (e.g., when your client needs to read a block of data from the server side jbod system). Both messages use the same format:

| **Bytes** | **Field** | **Description** |
|---|---|---|
| **0-1** | **length** | **The size of the packet in bytes** |
| **2-5** | **opcode** | **The opcode for the JBOD operation** |
| **6-7** | **return code** | **Return code from the JBOD operation** |
| **8-263** | **block** | **Where needed, a block of size JBOD BLOCK SIZE** |

In a nutshell, there are four steps.

(S1) the client side (inside the function jbod_client_operation) wraps all the parameters of a jbod operation into a JBOD request message and sends it as a packet to the server side;

(S2) the server receives the request message, extract the relevant fields (e.g., opcode, block if needed), issues the jbod_operation function to its local jbod system and receives the return code;

(S3) The server wraps the fields such as opcode, return code and block (if needed) into a JBOD response message and sends it to the client;

(S4) the client (inside the function jbod_client_operation) next receives the response message, extracts the relevant fields from it, and returns the return code and fill the parameter “block” if needed. Note that the first three fields (i.e., length, opcode and return code) of JBOD protocol messages can be considered as packet header, with the size HEADER LEN predefined in net.h. The block field can be considered as the optional payload. You can set the length field accordingly in the protocol messages to help the server infer whether a payload exists (the server side implementation follows the same logic).

**Implementation**

In addition to replacing all jbod_operation calls in mdadm.c with jbod_client_operation, you will implement functions defined in net.h in the provided net.c file. Specifically, you will implement jbod_connect function, which will connect to JBOD SERVER at port JBOD PORT, both defined in net.h, and jbod_disconnect function, which will close the connection to the JBOD server. Both of these functions will be called by the tester, not by your own code. The file net.c contains some functions with empty bodies that can help with structuring your code, but you may implement your own help functions as long as you implement those functions in net.h that will be directly called by tester.c and mdadm.c. That being said, following the structure would probably be the easiest way to debug/test/finish this project. Please refer to net.c for the detailed description on the purpose, parameters, and return value of each function.

**Testing**

We can test it by running the provided jbod_server in one terminal, which implements the server component of the protocol, and running the tester with the workload file in another terminal. Below is a sample session from the server and the client: Output from the jbod_server terminal:

```
        $ ./jbod_server JBOD server listening on port 3333... new client connection from
127.0.0.1 port 32402 client closed connection Output from the tester terminal:
        $ ./tester -w traces/random-input -s 1024 >x Cost: 17669400 Hit rate: 24.5%
        $ diff x traces/random-expected-output $
```

We can also run the jbod_server in verbose mode to print out every command that it receives from the client. Below is sample output that was trimmed to fit the space.

```
$ ./jbod_server -v

JBOD server listening on port 3333...

new client connection from 127.0.0.1 port 38546 received cmd id = 0 (JBOD_MOUNT) [disk id
= 0 block id = 0], result = 0
received cmd id = 2 (JBOD_SEEK_TO_DISK) [disk id = 0 block id = 0], result = 0
received cmd id = 5 (JBOD_WRITE_BLOCK) [disk id = 0 block id = 0], result = 0 block
contents:
```

```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

## Citation:

Author, Prof. Sencun Zhu & Prof. Suman Saha. (2022). *mdadm Linear Device* Department of Computer Science and Engineering, School of EECS, College of Engineering, The Pennsylvania State University.