Technical Report:

MDADM.C

Int global is used for mounting/unmounting which is set to -1 (i.e., unmounted)and int check basically checks if jbod operation causes error which is set to 0 (i.e., success for now).

1) `uint32_t op(int cmd,int disk_num, int block_num):`

   This is a wrapper function to compute the op code with the given parameters being shifted to its designated place as mentioned in the README pdf. We have command shifted by 26(i.e., JBOD_READ, JBOD_BLOCK,etc.) which specifies the command to be done. Then we have diskId (shifted by 22) and blockId which specifies the disk number and block number. Here, format brings everything together and then we return it.

2) `int mdadm_mount(void):`

   Here, we are checking if the jbod operation has the correct parameter of command, disk and block, Since, we do not have a buffer we have NULL as the other parameter. If check is incorrect then set the value of global to 1 and if not vice-versa.

3) `int mdadm_unmount(void):`

   Similarly, to mdadm_mount is mdadm_unmount. Here, we are checking if the jbod operation has the correct parameter of command (i.e., JBOD_UNMOUNT), disk and block, Since, we do not have a buffer we have NULL as the other parameter. If check is incorrect then set the value of global to as it is and if not then vice-versa.

4) `int min(int x, int y)`: This function is used to find the minimum number out of the two provided for offset later

5) `int mdadm_read(uint32_t addr, uint32_t len, uint8_t *buf)`:

   First, we are checking for the bounds like whether we are mounted, len is not out or below the respective bounds, buf is NULL or not.

   After that, we are doing the byte_counts for the bytes that we are done reading starting from 0. We are going to reading until our byte_count = len. For that we have parameters like diskId, blockId, offset, tempBuf (of 256) and bytes to read. We check to see if we are seeking to read across disk and seeking to block. After that, we check to see if we are reading across blocks. There is use of offset which specifies the exact location in the block/disk. After that we are using memcpy to copy where we are at everything so far with offset to the buf with the length of bytes to read. After doing all of that, we increment the byte count as we have it to do this until our count is same as len. At the same time, we increment our address as to move on.

6) `int mdadm_write`:

   As done in read, we also check for correct bounds before going any further.

   Similarly, we have byte_done_reading which keeps track of the numbers as well as to check seek to disk,block and read to block.

   Then, if we have to write a total less than block size(256), then we do that by seeking correctly, readin correctly and copying into memcpy to keep track of where we left of and then we write to the block. Now, if we have to write for anything greater than block size then we write for those who are less than block size and for the remaining we do it

in a loop until all the others have been written to. After that increment our parameters to move on to next value to finish.

CACHE.C

1) `int cache_create(int num_entries):`

   Check to see if the cache has been created. We also check the respective num_entries if they are above or below bounds. After that we allocate a specified amount (cache size) for our cache. Then we check if the cache has been created or not.

2) `int cache_destroy(void):`

   Similar to create, we check we have something or NULL as well as the parameters. In here, we free the cache that we have and set it to NULL and cache size to 0

3) `int cache_lookup(int disk_num, int block_num, uint8_t *buf):`

   First, we check for the relevant parameters. After that is f our cache is valid, correct disk number, correct block number then we use buffer to copy appropriate block to buf and keeps changing our clock, hits, and queries in order to have correct hit rate and requests track. p. If the lookup is successful, this function increments num_hits global variable; it also increments clock variable and assign it to the access_time field of the corresponding entry, to indicate that the entry was used recently.

4) `void cache_update(int disk_num, int block_num, const uint8_t *buf):`

In this function we keep updating our cache as we go along. For example, access time and block and keep tracks by the use of memcpy.

5) `int cache_insert(int disk_num, int block_num, const uint8_t *buf):`

   Here we insert the block identified by disk_num and block_num into the cache and copy buf to the respective cache entry. We are using least recently used policy states that an entry will be overwritten with insert function. This function increments and assigns clock variable to the access_time of the newly inserted entry. Here, initial block has the value of cache's ith element and keeps changing for every insert we do. So, basically we update the variables storing the initial value and the cache item accessed least recently.

6) `bool cache_enabled(void):`

   This just checks if cache is null or not and its size.

7) `void cache_print_hit_rate(void):`

   This function prints hit rate.

NET.C

Int cli_sd is set to -1

Bool check_value will check the value of calls as well as system call further

Int bytes_done is set to 0 as it will keep track when reading and writing

1) `static bool nread(int fd, int len, uint8_t *buf):`

The while loop runs until bytes done =len. Bytes_now is doing system call read and reading size which is len-bytes_done from sd into buf with the bytes done. Now, if the bytes_now areinvalid after that then return false and then increment bytes_done to bytes_now.

2) `static bool nwrite(int fd, int len, uint8_t *buf):`

Similar to nread, the while loop runs until bytes done =len. Bytes_now is doing system call write and writing size which is len-bytes_done from sd into buf with the bytes done. Now, if the bytes_now are invalid after that then return false and then increment bytes_done to bytes_now.

3) `static bool recv_packet(int fd, uint32_t *op, uint16_t *ret, uint8_t *block):`

First, we set the command to *op and shift it to appropriate bytes. Len_packet will either be 8 (header lenght) or 264 (header_len + block size). Now, if the command is to read block then len_packet will be 264 or else it will just be 8. opcode stores opcode of the packet and op stores converted opcode by using ntohl(). As for the return code if it is equal to 2, then the data is read from sd into the block with the length of block size

4) `static bool send_packet(int sd, uint32_t op, uint8_t *block):`

Similar to recv_packet are some of the variables like command, len_packet, header_len, blokc_size,etc. We are converting opcode to network bytes. As for memcpy, we are coping code to the packet+2 because of its location (according to Readme). If the command is to write bock then the return code is 2 and if the len of packet is greater

than 264 then we copy return code and block to the packet and then use check value to send packet to sd. Else we just copy returncode to packet and then send packet to sd but with len of 8 only.

5) `bool jbod_connect(const char *ip, uint16_t port):`

   First we call socket by specifying (its parameters) the socket type, address and protocol & if the cli_sd is -1 then we return false. After that we are using inet_aton() to convert the address to binary and store it to sin_addr. At the end, we call connect() with sockaddr_in and its respective parameters. If the connection is success then we return true else false.

6) `void jbod_disconnect(void):`

   In this we just close the socket and set it to -1

7) `int jbod_client_operation(uint32_t op, uint8_t *block):`

   We check if we are sending packets or not and if so then it is success (0) else failure (-1). Similarly, we check if we are receiving packets or not and if so then it is success (0) else failure (-1).