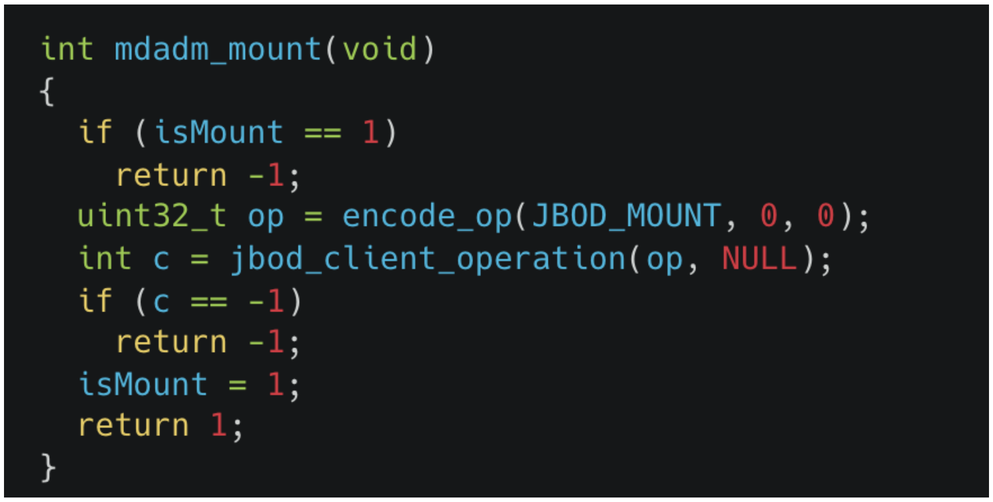
CMPSC 311 Assignment 5

Technical Report

In CMPSC 311 from assignment 2 to assignment 5, we have been dedicated to building a JBOD (Just a Bunch of Disks) library which can be read from, written to, cached and accessed remotely or locally. These functions were developed in the UNIX 64-bit architecture as it is the most efficient and lightweight operating system most efficient for managing such systems. It is used by large cloud providers like Microsoft Azure and AWS.

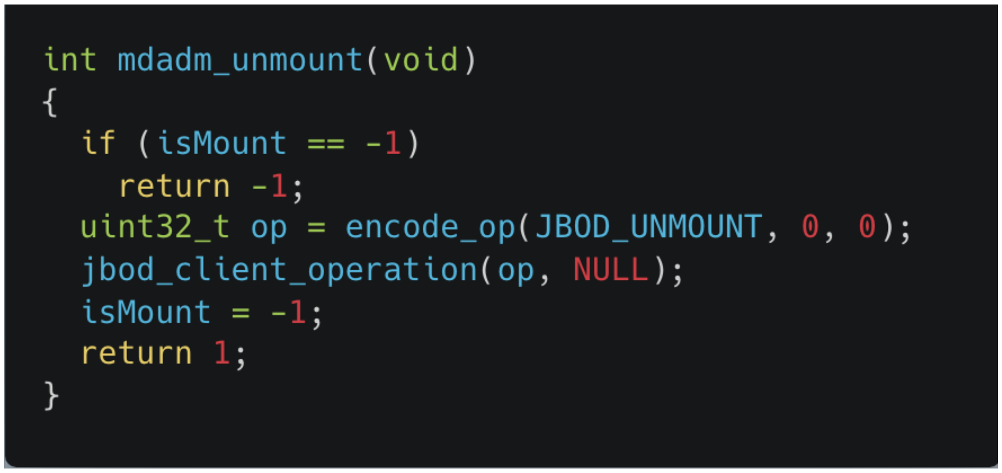
Let us start by describing the library. As apparent by its name, JBOD is a just a bunch of disks, in this case we have 16 disks, each disk has 256 blocks, and each block has a storage capacity of 256 bytes. The storage capacity of this library is 16*256*256 bytes ≈ 1 Megabyte(MB). To avoid comprising for speed this library also includes caching to help improve performance so that results can be accessed faster than the raw JBOD memory fetching and in the long run you might be surprised by how much money cache can save you. You can also access our library remotely if you are an authenticated user. All the functionality is available for remote use and there are no limitations. These functions have been developed using the native C language and libraries which to this day remains one of the few main programming languages for systems development. These functions have also been rigorously and thoroughly tested using various test cases and trace files which contain way more complicated data sets than day to day use. While using the library please note that a disk needs to mounted before it can be accessed, this can be done by using mdadm_mount(), you do not need to remember whether you mounted the disk before as the function does that for you. The maximum number of bytes which can be read from or written to the library is 1000 bytes. If at any point you try to exceed the amount the function will abort. The section below contains a brief description of all the functions you may use to start working with our library:

- `int mdadm_mount(void):` As the name suggest this function mounts the disks. This function must be called before accessing the disks remotely or locally as you cannot work on an unmounted library. This function returns 1 for success and -1 indicating failure.



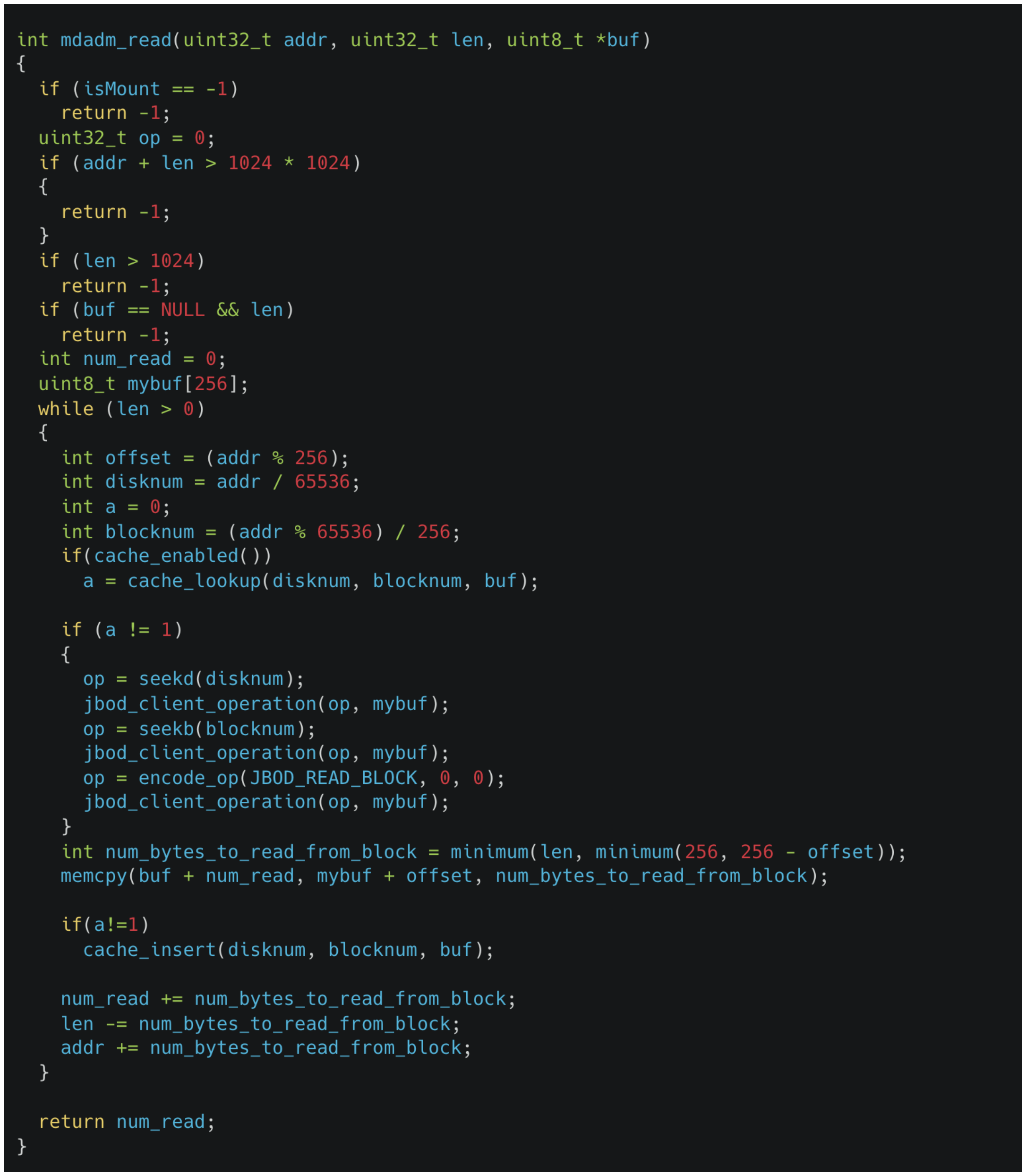
```
int mdadm_mount(void)
{
  if (isMount == 1)
    return -1;
  uint32_t op = encode_op(JBOD_MOUNT, 0, 0);
  int c = jbod_client_operation(op, NULL);
  if (c == -1)
    return -1;
  isMount = 1;
  return 1;
}
```

- `int mdadm_unmount(void):` Again, as the name suggests this function unmounts the disks. This function should be called after you are done using the library. This function returns 1 indicating success and -1 indicating failure.


```c
int mdadm_unmount(void)
{
  if (isMount == -1)
    return -1;
  uint32_t op = encode_op(JBOD_UNMOUNT, 0, 0);
  jbod_client_operation(op, NULL);
  isMount = -1;
  return 1;
}
```

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

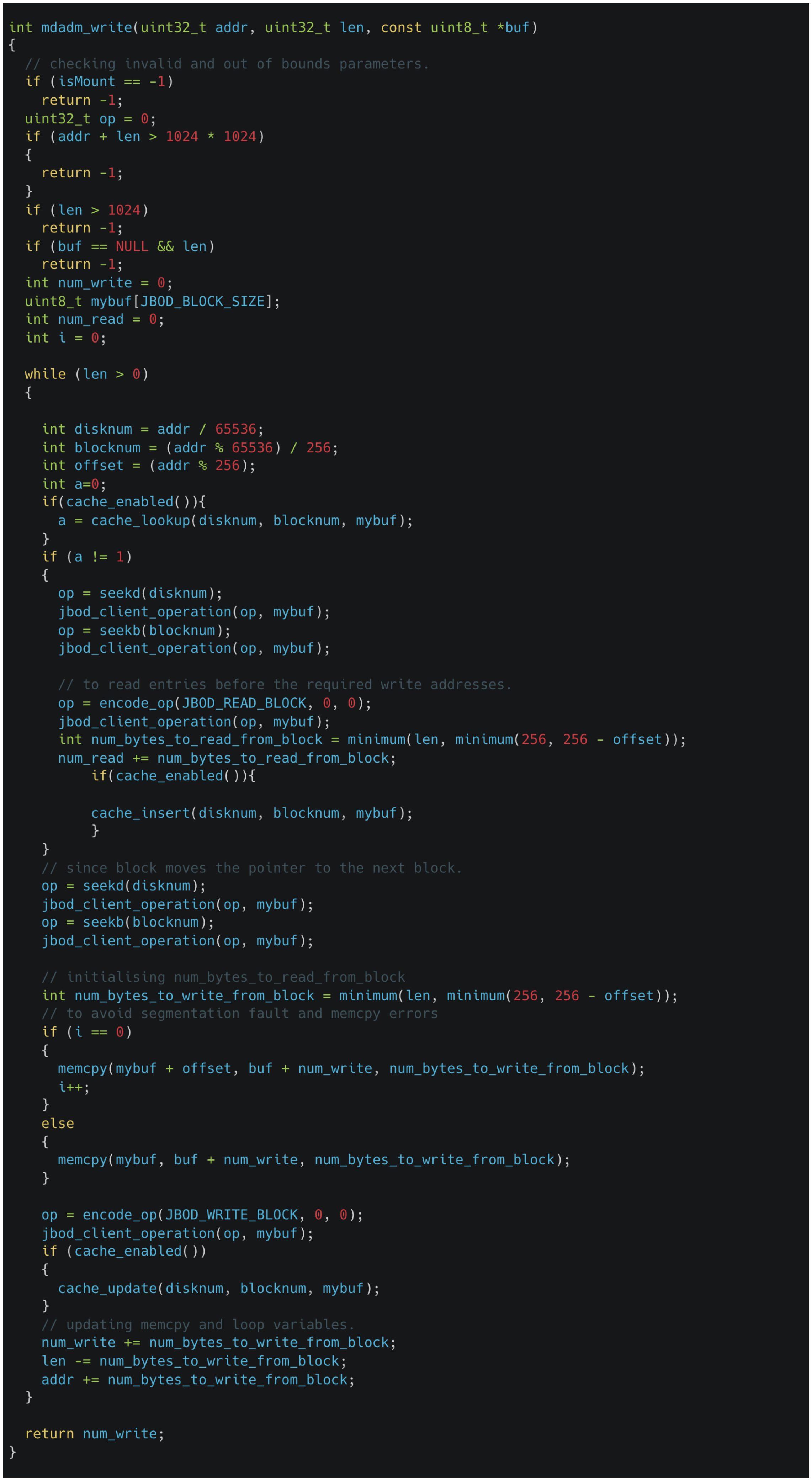
  This function allows the client to read stored content of len size at the given start address and store it in the buffer which points to an unsigned 8 bit integer. This function uses JBOD_READ_BLOCK operation which was predefined earlier. This function returns 1 indicating success and -1 indicating failure.



```
int mdadm_read(uint32_t addr, uint32_t len, uint8_t *buf)
{
  if (isMount == -1)
    return -1;
  uint32_t op = 0;
  if (addr + len > 1024 * 1024)
  {
    return -1;
  }
  if (len > 1024)
    return -1;
  if (buf == NULL && len)
    return -1;
  int num_read = 0;
  uint8_t mybuf[256];
  while (len > 0)
  {
    int offset = (addr % 256);
    int disknum = addr / 65536;
    int a = 0;
    int blocknum = (addr % 65536) / 256;
    if(cache_enabled())
      a = cache_lookup(disknum, blocknum, buf);

    if (a != 1)
    {
      op = seekd(disknum);
      jbod_client_operation(op, mybuf);
      op = seekb(blocknum);
      jbod_client_operation(op, mybuf);
      op = encode_op(JBOD_READ_BLOCK, 0, 0);
      jbod_client_operation(op, mybuf);
    }
    int num_bytes_to_read_from_block = minimum(len, minimum(256, 256 - offset));
    memcpy(buf + num_read, mybuf + offset, num_bytes_to_read_from_block);

    if(a!=1)
      cache_insert(disknum, blocknum, buf);

    num_read += num_bytes_to_read_from_block;
    len -= num_bytes_to_read_from_block;
    addr += num_bytes_to_read_from_block;
  }

  return num_read;
}
```

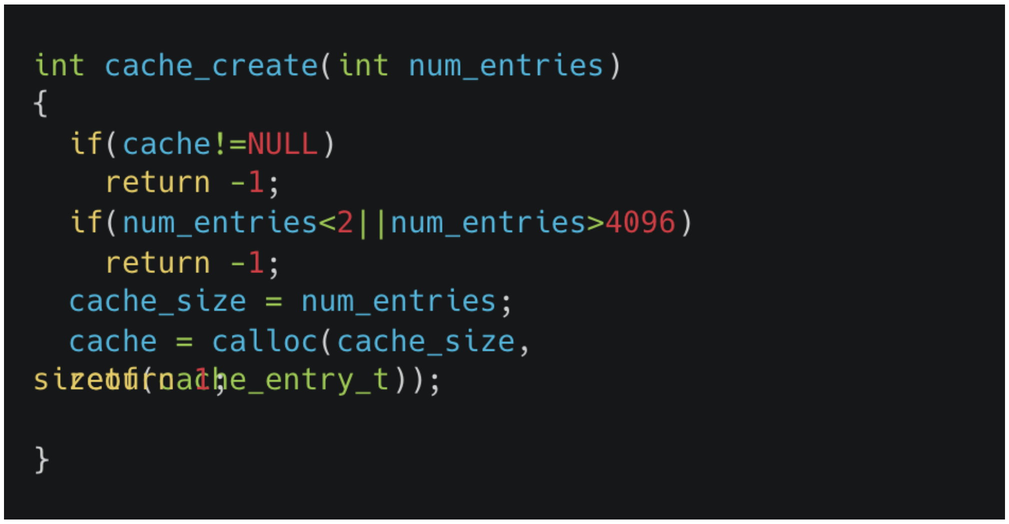
- `int mdadm_write(uint32_t addr, uint32_t len, uint8_t *buf):`

Like mdadm_read(), mdadm_write() takes the same inputs but reverses the operation, instead of reading to the buffer, it reads len bytes from the buffer to the given address. This function returns 1 indicating success and -1 indicating failure.



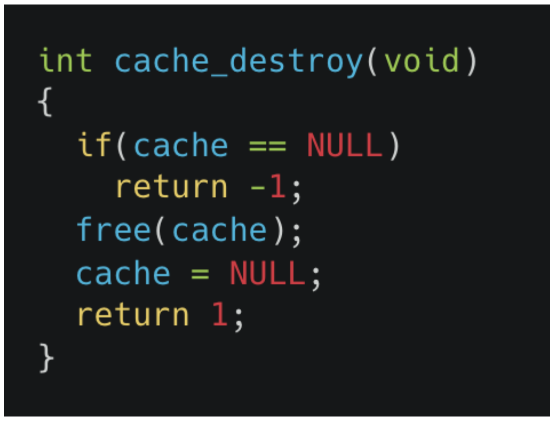
```c
int mdadm_write(uint32_t addr, uint32_t len, const uint8_t *buf)
{
  // checking invalid and out of bounds parameters.
  if (isMount == -1)
    return -1;
  uint32_t op = 0;
  if (addr + len > 1024 * 1024)
  {
    return -1;
  }
  if (len > 1024)
    return -1;
  if (buf == NULL && len)
    return -1;
  int num_write = 0;
  uint8_t mybuf[JBOD_BLOCK_SIZE];
  int num_read = 0;
  int i = 0;

  while (len > 0)
  {

    int disknum = addr / 65536;
    int blocknum = (addr % 65536) / 256;
    int offset = (addr % 256);
    int a=0;
    if(cache_enabled()){
      a = cache_lookup(disknum, blocknum, mybuf);
    }
    if (a != 1)
    {
      op = seekd(disknum);
      jbod_client_operation(op, mybuf);
      op = seekb(blocknum);
      jbod_client_operation(op, mybuf);

      // to read entries before the required write addresses.
      op = encode_op(JBOD_READ_BLOCK, 0, 0);
      jbod_client_operation(op, mybuf);
      int num_bytes_to_read_from_block = minimum(len, minimum(256, 256 - offset));
      num_read += num_bytes_to_read_from_block;
          if(cache_enabled()){

          cache_insert(disknum, blocknum, mybuf);
          }
    }
    // since block moves the pointer to the next block.
    op = seekd(disknum);
    jbod_client_operation(op, mybuf);
    op = seekb(blocknum);
    jbod_client_operation(op, mybuf);

    // initialising num_bytes_to_read_from_block
    int num_bytes_to_write_from_block = minimum(len, minimum(256, 256 - offset));
    // to avoid segmentation fault and memcpy errors
    if (i == 0)
    {
      memcpy(mybuf + offset, buf + num_write, num_bytes_to_write_from_block);
      i++;
    }
    else
    {
      memcpy(mybuf, buf + num_write, num_bytes_to_write_from_block);
    }

    op = encode_op(JBOD_WRITE_BLOCK, 0, 0);
    jbod_client_operation(op, mybuf);
    if (cache_enabled())
    {
      cache_update(disknum, blocknum, mybuf);
    }
    // updating memcpy and loop variables.
    num_write += num_bytes_to_write_from_block;
    len -= num_bytes_to_write_from_block;
    addr += num_bytes_to_write_from_block;
  }

  return num_write;
}
```

- `int cache_create(int num_entries):` This function allocates memory for use by cache. CAUTION: cache has a maximum size of 4096 bytes, a minimum size of 2 bytes, and cache must have content to be created. This function returns 1 indicating success and -1 indicating failure. Use this only if you want performance gain.



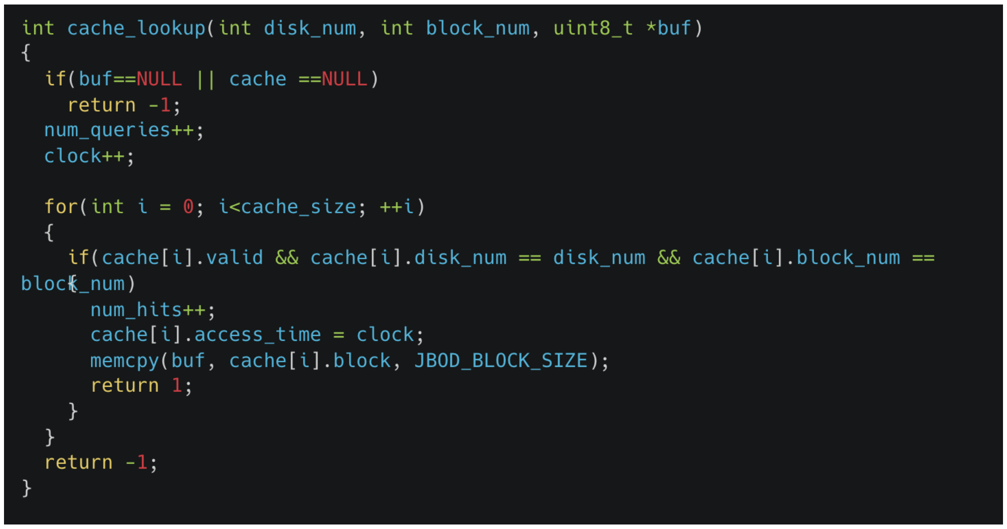
```
int cache_create(int num_entries)
{
  if(cache!=NULL)
    return -1;
  if(num_entries<2||num_entries>4096)
    return -1;
  cache_size = num_entries;
  cache = calloc(cache_size,
sizeof(cache_entry_t));
  return 1;

}
```

- `int cache_destroy(void):` This function frees allocated memory for cache, it must be called only when caching is no longer required. This function returns 1 indicating success and -1 indicating failure. Please implement this at the end to avoid memory leaks after multiple use.



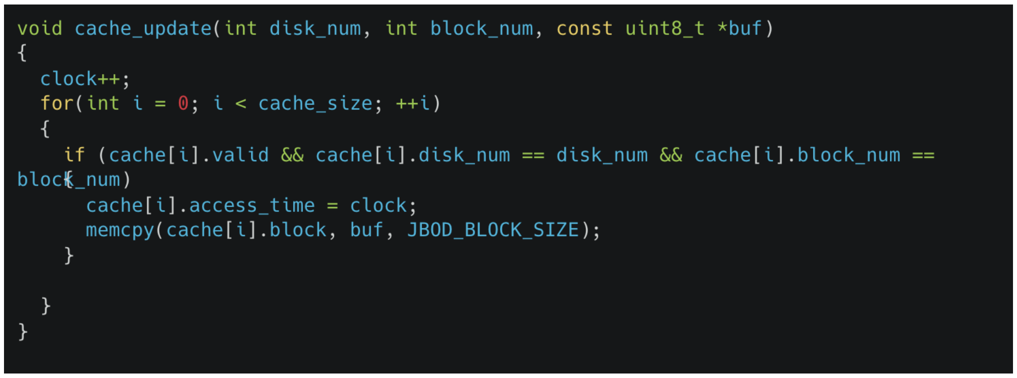
```
int cache_destroy(void)
{
  if(cache == NULL)
    return -1;
  free(cache);
  cache = NULL;
  return 1;
}
```

- `int cache_lookup(int disk_num, int block_num, uint8_t *buf):` This function looks up an element and a valid disk and block number and takes those as well as a buffer as input. The buffer and cache must be non-NULL for this function to work as intended. Caution: This time the disk_num and block_num are not for the library but for cache. This function looks up the element at the required address and copies it to the buffer. This function returns 1 indicating success and -1 indicating failure.



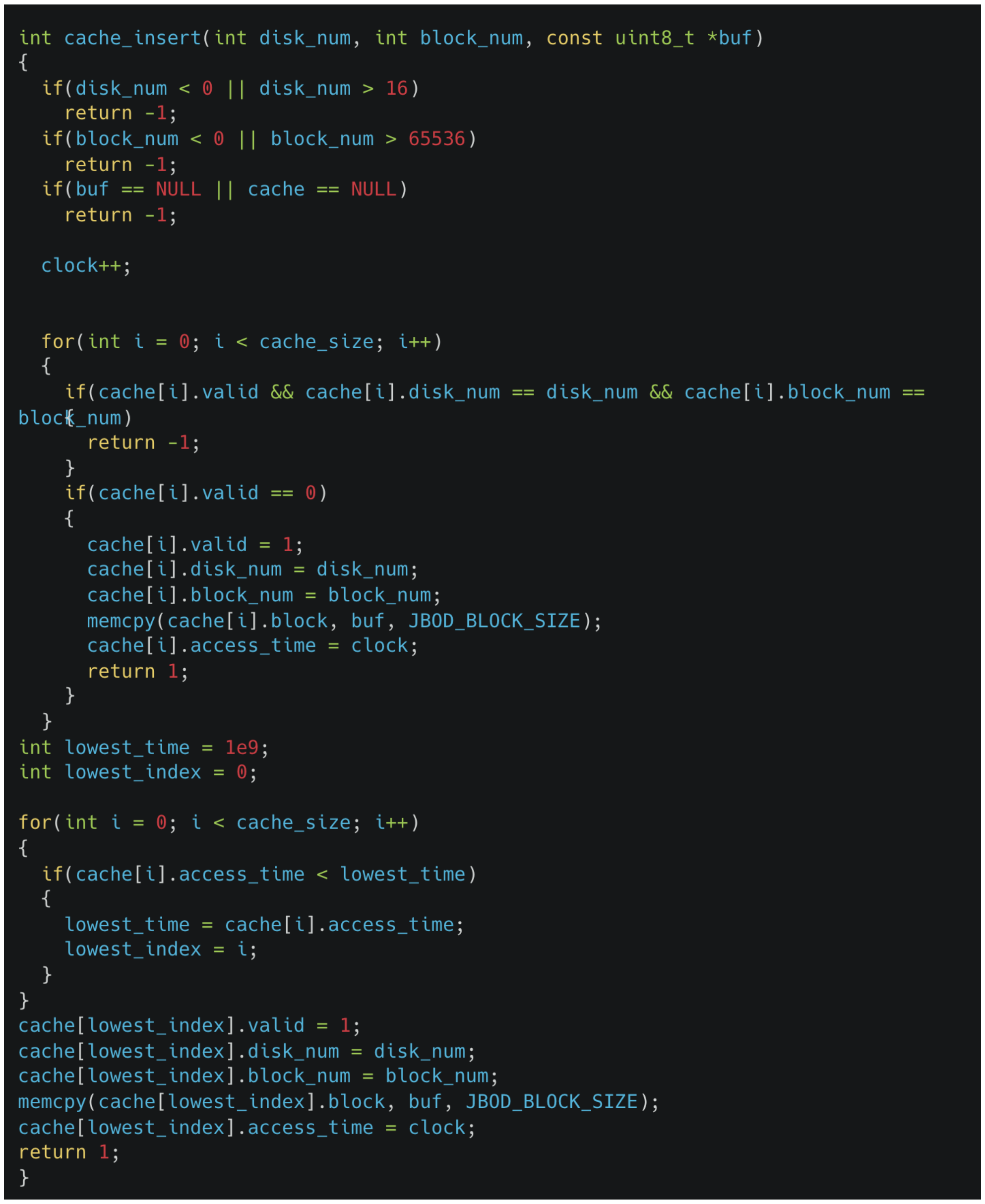
```
int cache_lookup(int disk_num, int block_num, uint8_t *buf)
{
  if(buf==NULL || cache ==NULL)
    return -1;
  num_queries++;
  clock++;

  for(int i = 0; i<cache_size; ++i)
  {
    if(cache[i].valid && cache[i].disk_num == disk_num && cache[i].block_num ==
block_num)
      num_hits++;
      cache[i].access_time = clock;
      memcpy(buf, cache[i].block, JBOD_BLOCK_SIZE);
      return 1;
    }
  }
  return -1;
}
```

- `void cache_update(int disk_num, int block_num, const uint8_t *buf):` This function updates a given disk and block address' contents with a new value contained in buffer. Caution: This time the disk_num and block_num are not for the library but for cache. This function gives no indication of success or failure, but you can always use lookup to check if the contents have been changed correctly.



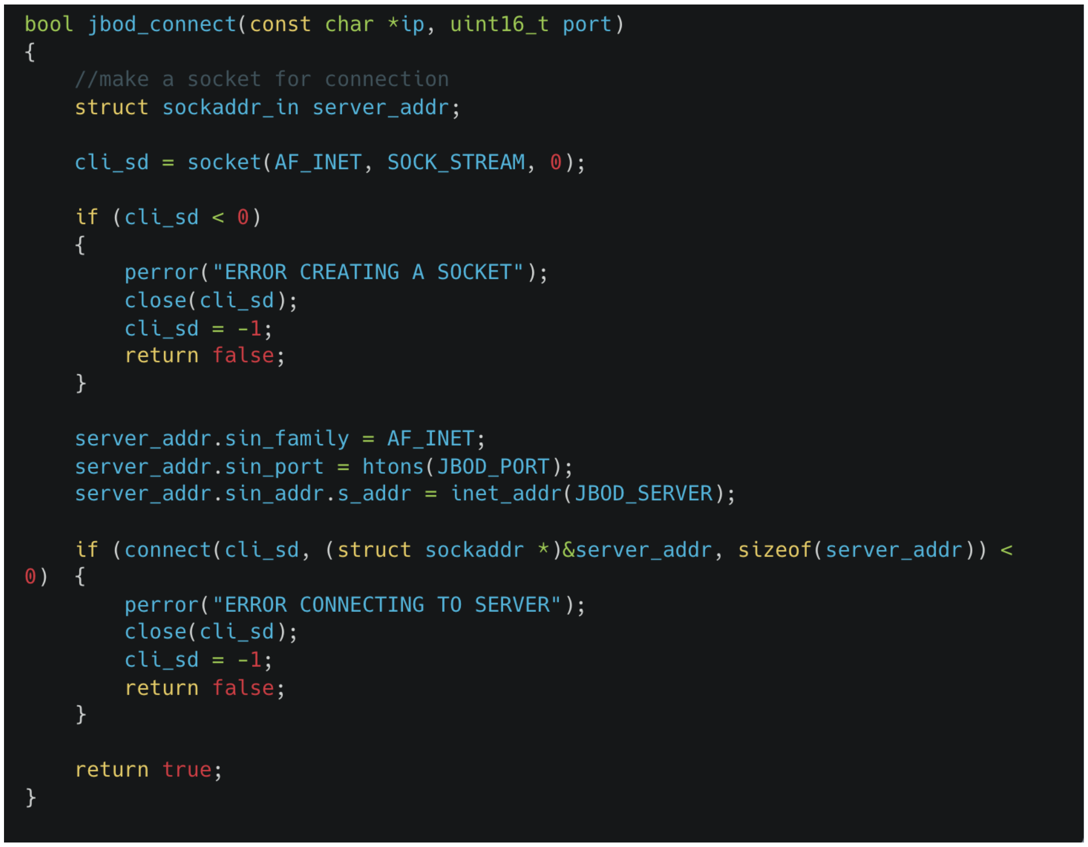
```
void cache_update(int disk_num, int block_num, const uint8_t *buf)
{
  clock++;
  for(int i = 0; i < cache_size; ++i)
  {
    if (cache[i].valid && cache[i].disk_num == disk_num && cache[i].block_num ==
block_num)
    {
      cache[i].access_time = clock;
      memcpy(cache[i].block, buf, JBOD_BLOCK_SIZE);
    }

  }
}
```

- `int cache_insert(int disk_num, int block_num, const uint8_t *buf):` This function inserts a set of values from the given buffer to the given block number in the cache. It checks for invalid parameters as indicated in the source code and then copies over the content using memcpy(). Caution: This time the disk_num and block_num are not for the library but for cache. This function returns 1 indicating success and -1 indicating failure.



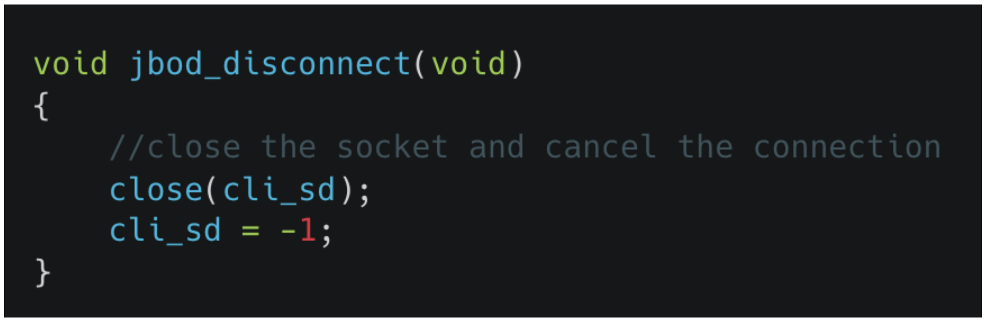
```
int cache_insert(int disk_num, int block_num, const uint8_t *buf)
{
  if(disk_num < 0 || disk_num > 16)
    return -1;
  if(block_num < 0 || block_num > 65536)
    return -1;
  if(buf == NULL || cache == NULL)
    return -1;

  clock++;


  for(int i = 0; i < cache_size; i++)
  {
    if(cache[i].valid && cache[i].disk_num == disk_num && cache[i].block_num ==
block_num)
      return -1;
    }
    if(cache[i].valid == 0)
    {
      cache[i].valid = 1;
      cache[i].disk_num = disk_num;
      cache[i].block_num = block_num;
      memcpy(cache[i].block, buf, JBOD_BLOCK_SIZE);
      cache[i].access_time = clock;
      return 1;
    }
  }
int lowest_time = 1e9;
int lowest_index = 0;

for(int i = 0; i < cache_size; i++)
{
  if(cache[i].access_time < lowest_time)
  {
    lowest_time = cache[i].access_time;
    lowest_index = i;
  }
}
cache[lowest_index].valid = 1;
cache[lowest_index].disk_num = disk_num;
cache[lowest_index].block_num = block_num;
memcpy(cache[lowest_index].block, buf, JBOD_BLOCK_SIZE);
cache[lowest_index].access_time = clock;
return 1;
}
```

- `bool jbod_connect(const char *ip, uint16_t port):` This function takes in the ip address and the port number as argument and then establishes a connection remotely to the library using a socket configured to the given ip(string) and port number. This function indicated true for success and false for failure. It is also indicated in the server terminal.



```c
bool jbod_connect(const char *ip, uint16_t port)
{
    //make a socket for connection
    struct sockaddr_in server_addr;

    cli_sd = socket(AF_INET, SOCK_STREAM, 0);

    if (cli_sd < 0)
    {
        perror("ERROR CREATING A SOCKET");
        close(cli_sd);
        cli_sd = -1;
        return false;
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(JBOD_PORT);
    server_addr.sin_addr.s_addr = inet_addr(JBOD_SERVER);

    if (connect(cli_sd, (struct sockaddr *)&server_addr, sizeof(server_addr)) <
0)  {
        perror("ERROR CONNECTING TO SERVER");
        close(cli_sd);
        cli_sd = -1;
        return false;
    }

    return true;
}
```

- `void jbod_disconnect(void):` This function disconnects the library from the server and terminates connection. Success/Failure is indicated in the server terminal.



```c
void jbod_disconnect(void)
{
    //close the socket and cancel the connection
    close(cli_sd);
    cli_sd = -1;
}
```

Please note that there are other functions which work in the background and have been abstracted so that you can read this documentation and start using our library as quickly as possible. It appears that this doc was written in some sort of markup/scripting language.

“But I haven't been consistent in my naming, because two years ago when I was designing the project, I wasn't thinking that we will ask you to write a manual on it. I'm not asking you to rename the functions, but I'm emphasizing this anyway for those of you who will go on to write their own libraries.” – Professor Abutalib Aghayev I will keep this in mind.

Note: I will reiterate that there are a few abstractions and confusing variables in the source code. For the full source code please check our open-source repository. Please check the invalid parameters in the source code of the functions in case I missed something. Since this is not a professional production build, there may be some trace errors which I may have missed. For software bugs and library maintenance, please contact: XXX-XXX-XXXX