Just a Bunch Of Disks (JBOD) Manual

Thomas Gutekunst

Table of Contents

Overview:

This program runs the read and write capabilities of mdadm from UNIX in the collection of disks know as JBOD. JBOD consists of 16 disks, each consisting of 256 blocks, and each block is made up of 256 bytes, resulting in JBOD containing 65,536 total bytes. Read and write allow the user to access data from JBOD and either alter it or just read it. LRU caching and networking were implemented as well to improve the efficiency and security of these operations.

mdadm.c

i. Description:

Firstly, mdadm stands for multiple disk and device administration, and it is a tool for doing cool tricks with multiple disks in Linux. This file will handle reading and writing into JBOD. There are 6 JBOD operations used in this file that are described below

ii. Functions

1. mdadm_mount()
2. mdadm_unmount()
3. mdadm_read()
4. mdadm_write()

iii. Index

1. int isMounted: keeps track if linear device is mounted.
2. jbod_client_operation(uint32_t, uint8_t*): Takes a JBOD operation and sends it over the jbod server network so that operations can be done within JBOD.
3. encode_op(int, int, int): takes an operation, disk number, and block number, and returns the encoded command so that it can be read by jbod_client_operation.
4. JBOD_MOUNT: Constant value for an operation that mounts all disks in JBOD and makes them ready to serve commands.

5. JBOD_UNMOUNT : Constant value for an operation that unmounts all disks in JBOD_SEEK_TO_DISK : Constant value for an operation that eeks 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 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.
6. JBOD_SEEK_TO_BLOCK: Constant value for an operation that 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.
7. JBOD_READ_BLOCK: Constant value for an operation that 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.

8. JBOD_WRITE_BLOCK: Constant value for an operation that 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.
9. memcpy(): built-in C function that copies the third parameter amount of bytes from the second parameter into the first parameter.

mdadm_mount():

i. Description

The mount function mounts all disks in the JBOD and makes them ready to serve commands. This is the first command that is 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.

ii. I/O

Input: void

Output: 1 on success, -1 on failure

iii. Function Flow:

1. Checks if disks are already mounted and fail if they are mounted.
2. Sets 32 bit unsigned int variable op to the encoded operation value of JBOD_MOUNT
3. Runs jbod_client_operation with op and NULL as the respective parameters
4. Sets Boolean global variable for checking if disks are mounted to true.
5. Returns 1

mdadm_unmount():

i. Description

The unmount function unmounts 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

ii. I/O

Input: void

Output: 1 on success, -1 on failure

iii. Function Flow:

1. Checks if disks are already unmounted and fail if they are unmounted.
2. Sets 32 bit unsigned int variable op to the encoded operation value of JBOD_UNMOUNT
3. Runs jbod_client_operation with op and NULL as the respective parameters
4. Sets Boolean global variable for checking if disks are mounted to false.
5. Returns 1

mdadm_read():

i. Description

Reads a user given amount of bytes into a buffer from JBOD starting at a user given address. The function fails if the linear device is not mounted. If the user gives an address larger than 1048574, then the function will fail due to it being an out-of-bound linear address, that being an address greater than what is contained in JBOD. Likewise, the function will fail if the length + the address is greater than 1048575 for the same reason. The reason length + address is given one extra bit than just address is because there can be an address of 1048574 if the length is only 1. The last two conditions for which the user's input will cause the function to fail are if the length is 1025 or greater and if the buffer in NULL.

ii. I/O

Input: addr: 32 bit unsigned int address of block being read, len: 32 bit unsigned int length of block being read, buf: pointer to an 8 bit unsigned int

Output: Return len on success and -1 on failure

iii. Function Flow:

1. Checks the parameters to make sure they are valid, and fail otherwise
2. Declares an int variable to keep track of how many bytes are read: num_read
3. Declares an int variable and sets it equal to len so that len can be edited but its value is still returned on success: templen

4. Calculates the disk and block number, as well as offset, of the starting address using addr. In this process it declares 2 32 bit unsigned int variables and one int variable for offset and sets the respective values to them: disknum, blocknum, offset.
5. Calls 2 seek operations using jbod_client_operation. The first is seek to disk with the parameters in jbod_client_operation being the encoded operation of JBOD_SEEK_TO_DISK with disknum and blocknum as the second and third parameters of encode_op, respectively, and the second parameter in jbod_client_operation is NULL. The second seek operation is the same as the first, but the first parameter in encode_op is JBOD_SEEK_TO_BLOCK rather than JBOD_SEEK_TO_WRITE. These operations set the internal I/O position to the user given address.
6. Declares an array of 32 bit unsigned ints to read into: mybuf
7. Read the block into mybuf by using jbod_client_operation with the first parameter being the encoded operation of JBOD_READ_BLOCK with zeros as the second and third parameter of encode_op, and the second jbod_client_operation parameter being mybuf.
8. Copies mybuf into buf using a series of memcpy’s. It does this by checking where in JBOD mybuf currently is and in what iteration. If it is the first iteration, checks if offset + len end within a block, and if so, copy len bytes into buf, but if it reads across the current block, copy 256 – offset into buf. If it is not the first iteration, check if the last block of the read is being read, and if it is, read len bytes into buf +

num_read, and if it isn’t the last block, read 256 bytes (the whole block) not buf + num_read.

9. Takes the number of bytes read and adds it to num_read and addr while subtracting it from len.
10. Repeats 4-9 until len = 0
11. Returns templen. The function could have returned len and edited templen rather than len, which is typically the standard way in coding, but this implementation was ultimately chosen.

mdadm_write():

i. Description

Writes a user given amount of bytes into JBOD from a buffer starting at a user given address. Contrary to read, the user given buffer is a constant, since it will not be edited, but written into JBOD. The conditions for which this function fails are almost exact to that of mdadm_read(), with two changes to the address and length + address parameter checks. It fails when the address given is greater than 1048575, or when the length + address is greater than 1045876. Every other parameter check is the same.

ii. I/O

Input: addr: 32 bit unsigned int address of block being read, len: 32 bit unsigned int length of block being read, buf: constant pointer to an 8 bit unsigned int

Output: Return len on success and -1 on failure

iii. Function Flow:

1. Checks the parameters to make sure they are valid, and fail otherwise
2. Declares an int variable to keep track of how many bytes are written: num_write
3. Declares an int variable and sets it equal to len so that len can be edited but its value is still returned on success: templen
4. Declares an array of 32 bit unsigned ints to write into: mybuf
5. Calculates the disk and block number, as well as offset, of the starting address using addr. In this process it declares 2 32 bit unsigned int variables and one int variable for offset and sets the respective values to them: disknum, blocknum, offset.

6. Calls 2 seek operations using jbod_client_operation. The first is seek to disk with the parameters in jbod_client_operation being the encoded operation of JBOD_SEEK_TO_DISK with disknum and blocknum as the second and third parameters of encode_op, respectively, and the second parameter in jbod_client_operation is NULL. The second seek operation is the same as the first, but the first parameter in encode_op is JBOD_SEEK_TO_BLOCK rather than JBOD_SEEK_TO_WRITE. These operations set the internal I/O position to the user given address.
7. Read the block into mybuf by using jbod_client_operation with the first parameter being the encoded operation of JBOD_READ_BLOCK with disknum and blocknum as the second and third parameters of encode_op, respectively, and the second jbod_client_operation parameter being mybuf.
8. Copies buf into mybuf using a series of memcpy's. It does this by checking where in JBOD mybuf currently is and in what iteration. If it is the first iteration, checks if offset + len end within a block, and if so, copy len bytes into mybuf + offset, but if it writes across the current block, copy 256 – offset into mybuf + offset. If it is not the first iteration, check if the last block of the write is being written, and if it is, write len bytes into mybuf, and if it isn't the last block, write 256 bytes (the whole block) into mybuf.
9. Takes the number of bytes read and adds it to num_read and addr while subtracting it from len.
10. Repeats 5-9 until len = 0

11. Repeat step 6.
12. Write mybuf into JBOD by calling jbod_client_operation with the first parameter being the encoded operation of JBOD_WRITE_BLOCK with disknum and blocknum as the second and third parameters of encode_op, respectively, and the second jbod_client_operation parameter being mybuf.
13. Returns templen. The function could have returned len and edited templen rather than len, which is typically the standard way in coding, but this implementation was ultimately chosen.

cache.c

i. Description:

cache.c implements LRU caching to mdadm.c. Instead of going to the disk every time to read or write in or from jbod, mdadm accesses the cache if it is enabled and contains the required data

ii. Functions

1. cache_create()
2. cache_destroy()

iii. Index

1. Cache_entry_t: type definition structure consisting of a bool, int, int, array of uint8_t of size 256, int. This is used to store an entry into the cache
2. cache: global variable of type pointer to a struct cache_entry_t. This variable essentially is the cache.
3. cache_size: global static variable of type int that stores the size of the cache in cache_create()
4. clock: global static variable of type int that is used to find the least recently used cache block.
5. num_queries: global static variable of type int that stores the amount of times the cache was accessed.

6. num_hits: global static variable of type int that stores the amount of hits in the cache.
7. cache_created: global variable of type bool that is used to check if the cache is already created
8. calloc(): built-in C function that dynamically allocates memory from the heap of a specific variable type for a certain amount of entries and size for all entries.
9. free(): built-in C function that frees the dynamically allocated memory

cache_create():

i. Description

Dynamically allocates space for a user given amount (num_entries) of cache entries and should store the address in the cache global variable. It also sets cache_size to num_entries, since that describes the size of the cache and will also be used by other functions. Calling this function twice without an intervening cache_destroy call will result in the function failing. It also fails if num_entries is less than 2 or greater than 4096. This function is needed to increase performance in caching and is highly recommended to be called by the user every time they want to use cache.

ii. I/O

Input: user given amount of cache entries: num_entries

Output: 1 on success and -1 on failure

iii. Function Flow:

1. Checks parameters and that a cache doesn't already exist.
2. Allocates memory from the heap to create the cache by calling the built-in C function, calloc, of type pointer to the struct cache_entry_t with each entry being the size od the struct cache_entry_t and there being num_entries entries.
3. Sets global variable cache_size to num_entries.
4. Sets global Boolean variable to check if a cache is created to true.
5. Returns 1.

cache_destroy():

i. Description

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. This function is needed to prevent memory leaks and is highly recommended to be called by the user every time they are finished using the cache to improve future performances..

ii. I/O

Input: void

Output: 1 on success and -1 on failure

iii. Function Flow:

1. Checks parameters and that a cache already exists.
2. Frees memory allocated from the heap by calling the built-in C function, free, on the global variable cache.
3. Sets global variable cache_size to 0.
4. Sets global variable cache to NULL
5. Sets global Boolean variable to check if a cache is created to false.
6. Returns 1.

net.c

i. Description:

net.c connects to a server to perform the 6 JBOD operations previously mentioned

ii. Functions

1. jbod_connect()
2. jbod_disconnect()

iii. Index

1. cli_sd: global variable of type int that functions as the client socket descriptor for the connection to the server.
2. JBOD_PORT: constant set to 3333.
3. AF_INET: built-in C constant that gives internet v4 protocol access.
4. SOCK_STREAM: built-in C function constant that defines a stream as the communication semantic.
5. htons(): built-in C function that converts a short value from host to network.
6. inet_aton(): built-in C function that converts an ipv4 into the UNIX structure used for processing.
7. socket(): built-in C function that creates a socket stream or datagram over a given domain.

8. connect(): built-in C function that connects the socket file descriptor to the specified address.
9. close(): built-in C function that closes the connection and deletes the associated entry in the operating system’s internal structures.

jbod_connect():

i. Description

Takes a user given ip and port and obtains an address for which a socket can be created and used to connect to the jbod server. Sets the global variable cli_sd to the socket. There are no input constraints.

ii. I/O

Input: ip address and port in variables of the same name

Output: true on success and false on failure

iii. Function Flow:

1. Declare variable of type struct sockaddr_in to use as client address when connecting to the server: client_address
2. Sets the fields of the server address by declaring two variables, client_address.sin_family and client_address.sin_port, and setting the first to AF_INET and the second to the host to network converted value of JBOD_PORT. The conversion is done while the value is being assigned by setting client_address.sin_port equal to htons(JBOD_PORT).
3. Converts the server address from ipv4 to network by calling inet_aton with parameters ip and the address of client_address.sin_addr. Fails if conversion returns 0.
4. Creates the socket by setting cli_sd to the socket function with parameters AF_INET, SOCK_STREAM, and 0. Fails if the socket function returns -1

5. Connects to the server byt calling the connect function with parameters cli_sd, the address of client_address, and size of client_address. Fails if connection returns -1
6. Returns true

jbod_disconnect():

i. Description

Disconnects from the server and resets cli_sd.

ii. I/O

Input: void

Output: void

iii. Function Flow:

1. Calls close on cli_sd.
2. Sets cli_sd to -1.
3. Returns.