mdadm Linear Device User Manual

Introduction - Basic Functionality

mdadm stands for Multiple Disk and Device Management. The mdadm Linear Device library is a C-based solution consisting of 8 functions that solves the problem of unifying multiple small disks under a unified storage system with a single address space. The library provides an easy-to-use interface for creating, managing, and using linear devices. It is a valuable resource for developers who need to consolidate storage, improve performance, or enhance security.

One good use of the mdadm Linear Device library is for companies that want to emphasize security by using multiple disks. By using the library to create a linear device, the company can make multiple disks appear as a single large disk to the operating system. This can make it easier to manage data across multiple disks and provide a higher level of security by distributing data across multiple physical devices. The user can call these functions listed below to mount, unmount, read and write to the disk:

- `mdadm_mount()`
- `mdadm_unmount()`
- `mdadm_read()`
- `mdadm_write()`

It also comes equipped with a block cache that helps in making the system more efficient and faster. This helps to improve the performance of the linear device and can be particularly useful in scenarios where high-speed access to data is critical, such as in database applications or high-performance computing environments. The cache can be created and destroyed with the following functions:

- `cache_create()`
- `cache_destroy()`

Another important feature of this library is its ability to connect to a JBOD server and execute JBOD operations over the network. This means that you can use it to manage multiple disks remotely, which can be particularly useful for companies with distributed teams or for situations where physical access to the disks is limited. Users can call the appropriate functions to connect and disconnect from the server:

- `jbod_connect()`
- `jbod_disconnect()`

Linear devices are a useful tool for a variety of tasks, including:

- Storage consolidation: It is used to consolidate multiple small disks into a single large disk. This can be useful for applications that require a large amount of storage space, such as database servers and media servers.
- Performance improvement: It is used to improve the performance of applications that access large amounts of data. This is because the data is spread across multiple disks, which can improve the read and write speeds.
- Security: It is used to improve the security of data by making it more difficult to steal or damage. This is because the data is spread across multiple disks, which makes it more difficult to access all of the data at once.

The mdadm Linear Device library is a versatile and powerful tool for managing storage across multiple disks. It can be used in a variety of situations to create virtual devices that are larger and more secure than individual physical devices. So, if you are looking for a tool to help you consolidate storage, improve performance, or enhance security, then the mdadm Linear Device library is a great option. It is a powerful and reliable tool that can be used to create and manage linear devices.

Mounting the disk

`int mdadm_mount(void);`

The `mdadm_mount()` function is used to mount the linear device, allowing users to read and perform operations on the combined address space of all disks. When this function is called, the linear device is mounted, providing a single address space that spans all disks.

If successful, the function returns a value of 1, indicating that the device has been successfully mounted. In case of failure, the function returns a value of -1.

It is important to note that calling `mdadm_mount()` function a second time without unmounting it in between will result in failure. This is because the function cannot mount the linear device again without first unmounting it. Therefore, it is necessary to call the `mdadm_unmount()` function to unmount the device before attempting to mount it again using `mdadm_mount().`

Unmounting the disk

`int mdadm_unmount(void);`

The `mdadm_unmount()` function is used to unmount the linear device, which in turn makes all commands to the device fail. It is important to note that attempting to call the `mdadm_unmount()` function for a linear device that has not been mounted using `mdadm_mount()` will result in a failure.

When the function is executed successfully, it returns a value of 1, whereas if it fails, it returns a value of -1.

Additionally, calling the `mdadm_unmount()` function more than once consecutively without first calling `mdadm_mount()` in between will also result in a failure.

Reading the disk

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

The `mdadm_read()` function is designed to read data from a specific memory address into a buffer of a specified length. The function takes three input parameters: `addr`, `len`, and `buf`. The `addr` parameter specifies the starting memory address from which the data is to be read. The `len` parameter specifies the length of data to be read and copied into the user-specified buffer `buf`.

`mdadm_read()` function returns an integer that represents the number of bytes read from the disk and copied into the provided buffer `buf`. If the function encounters any failure during the execution, it returns -1 as an error code to indicate that the read operation failed. Otherwise, it returns the number of bytes successfully read and copied into the buffer.

Several restrictions are imposed on the `mdadm_read()` function. First, attempting to read from an out-of-bound linear address will cause the function to fail. Second, the function will also fail if the length of data to be read exceeds 1,024 bytes. Additionally, if the `buf` buffer is NULL along with `len` being greater than zero, the function will also fail. The function also checks that the end of the linear address space is not exceeded by the requested read. Finally, the mount flag (to check if the disk is mounted) must be set to a non-zero value, i.e., the disk must be mounted, or the function will fail. These restrictions help prevent memory access errors that could lead to system crashes or other unpredictable behavior.

Writing to the disk

```
int mdadm_write(uint32_t addr, uint32_t len, const uint8_t *buf);
```

The `mdadm_write()` function writes `len` bytes from the user-supplied buf buffer to the storage system starting at address `addr`. The function takes three input parameters: `addr`, `len`, and `buf`. The `addr` parameter specifies the starting memory address from which the data is to be read. The `len` parameter specifies the length of data to be read and copied into the `buf` buffer.

`mdadm_write()` function returns the number of bytes successfully written to the storage system. If the function encounters any failures due to invalid parameters or an unmounted system, it will return -1 instead. Otherwise, the function will loop through the data to be written and write it to the storage system in chunks. Finally, the function returns the original length of data that was passed as an argument.

If `mdadm_write()` is called with a buffer that is out-of-bounds, the function will fail. Similarly, if the length of the write exceeds 1,024 bytes, the function will also fail. Other restrictions include checking whether `buf` is NULL when `len` is greater than zero, ensuring that `addr` is greater than or equal to zero and that (`addr+len`) is within the limits of the linear address space. Finally, if the storage system is not mounted, the function will fail.

Creating a cache

Adding a cache to mdadm system significantly improves its latency and reduced the load on the JBOD. The key is tuple consisting of disk number and block number that identifies a specific block in JBOD, and the value is contents of the block. When the users of mdadm system issue `mdadm_read()` call, 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 it is read from JBOD and inserted to the cache, so that if a user asks for the block again, it can be served faster from the cache.

```
int cache_create(int num_entries);
```

The `cache_create()` function is responsible for dynamically allocating space for a specified number of cache entries and storing the address in the cache global variable. This function takes an integer argument `num_entries`, which specifies the number of cache entries to allocate.

The `cache_create()` function returns an integer value indicating the status of the operation. If the function successfully creates the cache, it returns a value of 1. If the function fails due to an invalid value of `num_entries`, it returns a value of -1. It is important to note that calling the `cache_create()` function twice without an intervening call to the `cache_destroy()` function will result in failure. If the value of `num_entries` is less than 2 or greater than 4096, the function will fail. The `cache_create()` function is an essential part of the caching mechanism in software systems.

Destroying a cache

```
int cache_destroy(void);
```

The `cache_destroy()` function is responsible for freeing the dynamically allocated space for the cache and setting the cache variable to NULL. The function takes no arguments, and its operation depends on the state of the cache. If the cache has not been initialized, the function does nothing.

The `cache_destroy()` function returns an integer value indicating the status of the operation. If the function successfully destroys the cache, it returns a value of 1. If the function fails due to the cache not being initialized, it returns a value of -1.

It is important to note that calling the `cache_destroy()` function twice without an intervening call to the `cache_create()` function will result in failure. The `num_entries` argument can have a minimum value of 2 and a maximum value of 4096. The `cache_destroy()` function ensures that memory resources are efficiently managed and prevents memory leaks in software systems.

Connecting to JBOD server

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

The `jbod_connect()` function is designed to establish a connection between the client and the JBOD server. The function takes two parameters: the IP address of the server and the port number to be used for communication. The function uses a socket-based approach to establish the connection between the client and the server. Having networking support in mdadm allows avoidance of downtime in case a JBOD system malfunctions, by switching to another JBOD system on the fly. It creates a socket using the `socket()` function, specifying the address family, socket type, and protocol. If the socket creation is unsuccessful, the function returns false. The `jbod_connect()` function is a crucial component of any system that uses JBOD devices, as it allows the client to communicate with the server and perform various operations on the JBOD devices, such as data storage and retrieval.

Disconnecting to JBOD server

```
void jbod_disconnect(void);
```

The `jbod_disconnect()` function is designed to terminate the connection between the client and the JBOD server. The function takes no arguments and simply closes the socket connection by calling the `close()` function. This function is an essential component of any JBOD-based system, as it ensures that the client does not remain connected to the server unnecessarily and avoids any potential issues related to resource allocation and memory leaks.