NAME

md − Multiple Device driver aka Linux Software RAID

SYNOPSIS

/dev/md*n*
/dev/md/*n*
/dev/md/name

DESCRIPTION

The **md** driver provides virtual devices that are created from one or more independent underlying devices. This array of devices often contains redundancy and the devices are often disk drives, hence the acronym RAID which stands for a Redundant Array of Independent Disks.

md supports RAID levels 1 (mirroring), 4 (striped array with parity device), 5 (striped array with distributed parity information), 6 (striped array with distributed dual redundancy information), and 10 (striped and mirrored). If some number of underlying devices fails while using one of these levels, the array will continue to function; this number is one for RAID levels 4 and 5, two for RAID level 6, and all but one (N-1) for RAID level 1, and dependent on configuration for level 10.

md also supports a number of pseudo RAID (non-redundant) configurations including RAID0 (striped array), LINEAR (catenated array), MULTIPATH (a set of different interfaces to the same device), and FAULTY (a layer over a single device into which errors can be injected).

MD METADATA

Each device in an array may have some *metadata* stored in the device. This metadata is sometimes called a **superblock**. The metadata records information about the structure and state of the array. This allows the array to be reliably re-assembled after a shutdown.

From Linux kernel version 2.6.10, **md** provides support for two different formats of metadata, and other formats can be added. Prior to this release, only one format is supported.

The common format — known as version 0.90 — has a superblock that is 4K long and is written into a 64K aligned block that starts at least 64K and less than 128K from the end of the device (i.e. to get the address of the superblock round the size of the device down to a multiple of 64K and then subtract 64K). The available size of each device is the amount of space before the super block, so between 64K and 128K is lost when a device in incorporated into an MD array. This superblock stores multi-byte fields in a processor-dependent manner, so arrays cannot easily be moved between computers with different processors.

The new format — known as version 1 — has a superblock that is normally 1K long, but can be longer. It is normally stored between 8K and 12K from the end of the device, on a 4K boundary, though variations can be stored at the start of the device (version 1.1) or 4K from the start of the device (version 1.2). This metadata format stores multibyte data in a processor-independent format and supports up to hundreds of component devices (version 0.90 only supports 28).

The metadata contains, among other things:

LEVEL
: The manner in which the devices are arranged into the array (LINEAR, RAID0, RAID1, RAID4, RAID5, RAID10, MULTIPATH).

UUID
: a 128 bit Universally Unique Identifier that identifies the array that contains this device.

When a version 0.90 array is being reshaped (e.g. adding extra devices to a RAID5), the version number is temporarily set to 0.91. This ensures that if the reshape process is stopped in the middle (e.g. by a system crash) and the machine boots into an older kernel that does not support reshaping, then the array will not be assembled (which would cause data corruption) but will be left untouched until a kernel that can complete the reshape processes is used.

ARRAYS WITHOUT METADATA

While it is usually best to create arrays with superblocks so that they can be assembled reliably, there are some circumstances when an array without superblocks is preferred. These include:

LEGACY ARRAYS
: Early versions of the **md** driver only supported LINEAR and RAID0 configurations and did not use a superblock (which is less critical with these configurations). While such arrays should be rebuilt with superblocks if possible, **md** continues to support them.

FAULTY
: Being a largely transparent layer over a different device, the FAULTY personality doesn't gain anything from having a superblock.

MULTIPATH
: It is often possible to detect devices which are different paths to the same storage directly rather than having a distinctive superblock written to the device and searched for on all paths. In this case, a MULTIPATH array with no superblock makes sense.

RAID1
: In some configurations it might be desired to create a RAID1 configuration that does not use a superblock, and to maintain the state of the array elsewhere. While not encouraged for general use, it does have special-purpose uses and is supported.

ARRAYS WITH EXTERNAL METADATA

From release 2.6.28, the *md* driver supports arrays with externally managed metadata. That is, the metadata is not managed by the kernel but rather by a user-space program which is external to the kernel. This allows support for a variety of metadata formats without cluttering the kernel with lots of details.

md is able to communicate with the user-space program through various sysfs attributes so that it can make appropriate changes to the metadata − for example to mark a device as faulty. When necessary, *md* will wait for the program to acknowledge the event by writing to a sysfs attribute. The manual page for*mdmon*(8) contains more detail about this interaction.

CONTAINERS

Many metadata formats use a single block of metadata to describe a number of different arrays which all use the same set of devices. In this case it is helpful for the kernel to know about the full set of devices as a whole. This set is known to md as a *container*. A container is an*md* array with e xternally managed metadata and with device offset and size so that it just covers the metadata part of the devices. The remainder of each device is available to be incorporated into various arrays.

LINEAR

A LINEAR array simply catenates the available space on each drive to form one large virtual drive.

One advantage of this arrangement over the more common RAID0 arrangement is that the array may be reconfigured at a later time with an extra drive, so the array is made bigger without disturbing the data that is on the array. This can even be done on a live array.

If a chunksize is given with a LINEAR array, the usable space on each device is rounded down to a multiple of this chunksize.

RAID0

A RAID0 array (which has zero redundancy) is also known as a striped array. A RAID0 array is configured at creation with a **Chunk Size** which must be a power of two (prior to Linux 2.6.31), and at least 4 kibibytes.

The RAID0 driver assigns the first chunk of the array to the first device, the second chunk to the second device, and so on until all drives have been assigned one chunk. This collection of chunks forms a **stripe**.

Further chunks are gathered into stripes in the same way, and are assigned to the remaining space in the drives.

If devices in the array are not all the same size, then once the smallest device has been exhausted, the RAID0 driver starts collecting chunks into smaller stripes that only span the drives which still have remaining space.

A bug was introduced in linux 3.14 which changed the layout of blocks in a RAID0 beyond the region that is striped over all devices. This bug does not affect an array with all devices the same size, but can affect other RAID0 arrays.

Linux 5.4 (and some stable kernels to which the change was backported) will not normally assemble such an array as it cannot know which layout to use. There is a module parameter "raid0.default_layout" which can be set to "1" to force the kernel to use the pre-3.14 layout or to "2" to force it to use the 3.14-and-later layout. when creating a new RAID0 array, *mdadm* will record the chosen layout in the metadata in a way that allows newer kernels to assemble the array without needing a module parameter.

To assemble an old array on a new kernel without using the module parameter, use either the **--update=layout-original** option or the **--update=layout-alternate** option.

Once you have updated the layout you will not be able to mount the array on an older kernel. If you need to revert to an older kernel, the layout information can be erased with the **--update=layout-unspecificed** option. If you use this option to**--assemble** while running a ne wer kernel, the array will NOT assemble, but the metadata will be update so that it can be assembled on an older kernel.

No that setting the layout to "unspecified" removes protections against this bug, and you must be sure that the kernel you use matches the layout of the array.

RAID1

A RAID1 array is also known as a mirrored set (though mirrors tend to provide reflected images, which RAID1 does not) or a plex.

Once initialised, each device in a RAID1 array contains exactly the same data. Changes are written to all devices in parallel. Data is read from any one device. The driver attempts to distribute read requests across all devices to maximise performance.

All devices in a RAID1 array should be the same size. If they are not, then only the amount of space available on the smallest device is used (any extra space on other devices is wasted).

Note that the read balancing done by the driver does not make the RAID1 performance profile be the same as for RAID0; a single stream of sequential input will not be accelerated (e.g. a single dd), but multiple sequential streams or a random workload will use more than one spindle. In theory, having an N-disk RAID1 will allow N sequential threads to read from all disks.

Individual devices in a RAID1 can be marked as "write-mostly". These drives are excluded from the normal read balancing and will only be read from when there is no other option. This can be useful for devices connected over a slow link.

RAID4

A RAID4 array is like a RAID0 array with an extra device for storing parity. This device is the last of the active devices in the array. Unlike RAID0, RAID4 also requires that all stripes span all drives, so extra space on devices that are larger than the smallest is wasted.

When any block in a RAID4 array is modified, the parity block for that stripe (i.e. the block in the parity

device at the same device offset as the stripe) is also modified so that the parity block always contains the "parity" for the whole stripe. I.e. its content is equivalent to the result of performing an exclusive-or operation between all the data blocks in the stripe.

This allows the array to continue to function if one device fails. The data that was on that device can be calculated as needed from the parity block and the other data blocks.

RAID5

RAID5 is very similar to RAID4. The difference is that the parity blocks for each stripe, instead of being on a single device, are distributed across all devices. This allows more parallelism when writing, as two different block updates will quite possibly affect parity blocks on different devices so there is less contention.

This also allows more parallelism when reading, as read requests are distributed over all the devices in the array instead of all but one.

RAID6

RAID6 is similar to RAID5, but can handle the loss of any *two* devices without data loss. Accordingly, it requires N+2 drives to store N drives worth of data.

The performance for RAID6 is slightly lower but comparable to RAID5 in normal mode and single disk failure mode. It is very slow in dual disk failure mode, however.

RAID10

RAID10 provides a combination of RAID1 and RAID0, and is sometimes known as RAID1+0. Every datablock is duplicated some number of times, and the resulting collection of datablocks are distributed over multiple drives.

When configuring a RAID10 array, it is necessary to specify the number of replicas of each data block that are required (this will usually be 2) and whether their layout should be "near", "far" or "offset" (with "offset" being available since Linux 2.6.18).

About the RAID10 Layout Examples:
The examples below visualise the chunk distribution on the underlying devices for the respective layout.

For simplicity it is assumed that the size of the chunks equals the size of the blocks of the underlying devices as well as those of the RAID10 device exported by the kernel (for example **/dev/md/**name).
Therefore the chunks / chunk numbers map directly to the blocks /block addresses of the exported RAID10 device.

Decimal numbers (0, 1, 2, ...) are the chunks of the RAID10 and due to the above assumption also the blocks and block addresses of the exported RAID10 device.
Repeated numbers mean copies of a chunk / block (obviously on different underlying devices).
Hexadecimal numbers (0x00, 0x01, 0x02, ...) are the block addresses of the underlying devices.

"near" Layout

When "near" replicas are chosen, the multiple copies of a given chunk are laid out consecutively ("as close to each other as possible") across the stripes of the array.

With an even number of devices, they will likely (unless some misalignment is present) lay at the very same offset on the different devices.
This is as the "classic" RAID1+0; that is two groups of mirrored devices (in the example below the groups Device #1 / #2 and Device #3 / #4 are each a RAID1) both in turn forming a striped

RAID0.

Example with 2 copies per chunk and an even number (4) of devices:

	Device #1	Device #2	Device #3	Device #4
0x00	0	0	1	1
0x01	2	2	3	3
	...	...	...	...
:	:	:	:	:
	...	...	...	...
0x80	254	254	255	255

\---------v---------/ \---------v---------/
RAID1 RAID1
\----------v----------/
RAID0

Example with 2 copies per chunk and an odd number (5) of devices:

	Dev #1	Dev #2	Dev #3	Dev #4	Dev #5
0x00	0	0	1	1	2
0x01	2	3	3	4	4
	...	...	...	...	...
:	:	:	:	:	:
	...	...	...	...	...
0x80	317	318	318	319	319

"far" Layout

When "far" replicas are chosen, the multiple copies of a given chunk are laid out quite distant ("as far as reasonably possible") from each other.

First a complete sequence of all data blocks (that is all the data one sees on the exported RAID10 block device) is striped over the devices. Then another (though "shifted") complete sequence of all data blocks; and so on (in the case of more than 2 copies per chunk).

The "shift" needed to prevent placing copies of the same chunks on the same devices is actually a cyclic permutation with offset 1 of each of the stripes within a complete sequence of chunks.
The offset 1 is relative to the previous complete sequence of chunks, so in case of more than 2 copies per chunk one gets the following offsets:
1. complete sequence of chunks: offset = 0
2. complete sequence of chunks: offset = 1
3. complete sequence of chunks: offset = 2
:
n. complete sequence of chunks: offset = n-1

Example with 2 copies per chunk and an even number (4) of devices:

	Device #1	Device #2	Device #3	Device #4	
0x00	0	1	2	3	\
0x01	4	5	6	7	> [#]
	...	...	...	...	...
:	:	:	:	:	:
	...	...	...	...	...
0x40	252	253	254	255	/
0x41	3	0	1	2	\
0x42	7	4	5	6	> [#]˜
	...	...	...	...	...
:	:	:	:	:	:
	...	...	...	...	...
0x80	255	252	253	254	/

Example with 2 copies per chunk and an odd number (5) of devices:

	Dev #1	Dev #2	Dev #3	Dev #4	Dev #5	
0x00	0	1	2	3	4	\
0x01	5	6	7	8	9	> [#]
	...	...	...	...	...	...
:	:	:	:	:	:	:
	...	...	...	...	...	...
0x40	315	316	317	318	319	/
0x41	4	0	1	2	3	\
0x42	9	5	6	7	8	> [#]˜
	...	...	...	...	...	...
:	:	:	:	:	:	:
	...	...	...	...	...	...
0x80	319	315	316	317	318	/

With [#] being the complete sequence of chunks and [#]˜ the cyclic permutation with offset 1 thereof (in the case of more than 2 copies per chunk there would be ([#]˜)˜, (([#]˜)˜)˜, ...).

The advantage of this layout is that MD can easily spread sequential reads over the devices, making them similar to RAID0 in terms of speed.
The cost is more seeking for writes, making them substantially slower.

"offset" Layout

When "offset" replicas are chosen, all the copies of a given chunk are striped consecutively ("offset by the stripe length after each other") over the devices.

Explained in detail, <number of devices> consecutive chunks are striped over the devices, immediately followed by a "shifted" copy of these chunks (and by further such "shifted" copies in the case of more than 2 copies per chunk).
This pattern repeats for all further consecutive chunks of the exported RAID10 device (in other words: all further data blocks).

The "shift" needed to prevent placing copies of the same chunks on the same devices is actually a cyclic permutation with offset 1 of each of the striped copies of <number of devices> consecutive chunks.
The offset 1 is relative to the previous striped copy of <number of devices> consecutive chunks, so in case of more than 2 copies per chunk one gets the following offsets:

1. <number of devices> consecutive chunks: offset = 0
2. <number of devices> consecutive chunks: offset = 1
3. <number of devices> consecutive chunks: offset = 2
:
n. <number of devices> consecutive chunks: offset = n-1

Example with 2 copies per chunk and an even number (4) of devices:

	Device #1	Device #2	Device #3	Device #4	
0x00	0	1	2	3	) AA
0x01	3	0	1	2	) AA˜
0x02	4	5	6	7	) AB
0x03	7	4	5	6	) AB˜
	...	...	...	...	...
:	:	:	:	:	:
	...	...	...	...	...
0x79	251	252	253	254	) EX
0x80	254	251	252	253	) EX˜

Example with 2 copies per chunk and an odd number (5) of devices:

	Dev #1	Dev #2	Dev #3	Dev #4	Dev #5	
0x00	0	1	2	3	4	) AA
0x01	4	0	1	2	3	) AA˜
0x02	5	6	7	8	9	) AB
0x03	9	5	6	7	8	) AB˜
	...	...	...	...	...	...
:	:	:	:	:	:	:
	...	...	...	...	...	...
0x79	314	315	316	317	318	) EX
0x80	318	314	315	316	317	) EX˜

With AA, AB, ..., AZ, BA, ... being the sets of <number of devices> consecutive chunks and AA˜, AB˜, ..., AZ˜, BA˜, ... the cyclic permutations with offset 1 thereof (in the case of more than 2 copies per chunk there would be (AA˜)˜, ... as well as ((AA˜)˜)˜, ... and so on).

This should give similar read characteristics to "far" if a suitably large chunk size is used, but without as much seeking for writes.

It should be noted that the number of devices in a RAID10 array need not be a multiple of the number of replica of each data block; however, there must be at least as many devices as replicas.

If, for example, an array is created with 5 devices and 2 replicas, then space equivalent to 2.5 of the devices will be available, and every block will be stored on two different devices.

Finally, it is possible to have an array with both "near" and "far" copies. If an array is configured with 2 near copies and 2 far copies, then there will be a total of 4 copies of each block, each on a different drive. This is an artifact of the implementation and is unlikely to be of real value.

MULTIPATH

MULTIPATH is not really a RAID at all as there is only one real device in a MULTIPATH md array. However there are multiple access points (paths) to this device, and one of these paths might fail, so there are some similarities.

A MULTIPATH array is composed of a number of logically different devices, often fibre channel interfaces,

that all refer the the same real device. If one of these interfaces fails (e.g. due to cable problems), the MULTIPATH driver will attempt to redirect requests to another interface.

The MULTIPATH drive is not receiving any ongoing development and should be considered a legacy driver. The device-mapper based multipath drivers should be preferred for new installations.

FAULTY

The FAULTY md module is provided for testing purposes. A FAULTY array has exactly one component device and is normally assembled without a superblock, so the md array created provides direct access to all of the data in the component device.

The FAULTY module may be requested to simulate faults to allow testing of other md levels or of filesystems. Faults can be chosen to trigger on read requests or write requests, and can be transient (a subsequent read/write at the address will probably succeed) or persistent (subsequent read/write of the same address will fail). Further, read faults can be "fixable" meaning that they persist until a write request at the same address.

Fault types can be requested with a period. In this case, the fault will recur repeatedly after the given number of requests of the relevant type. For example if persistent read faults have a period of 100, then every 100th read request would generate a fault, and the faulty sector would be recorded so that subsequent reads on that sector would also fail.

There is a limit to the number of faulty sectors that are remembered. Faults generated after this limit is exhausted are treated as transient.

The list of faulty sectors can be flushed, and the active list of failure modes can be cleared.

UNCLEAN SHUTDOWN

When changes are made to a RAID1, RAID4, RAID5, RAID6, or RAID10 array there is a possibility of inconsistency for short periods of time as each update requires at least two block to be written to different devices, and these writes probably won't happen at exactly the same time. Thus if a system with one of these arrays is shutdown in the middle of a write operation (e.g. due to power failure), the array may not be consistent.

To handle this situation, the md driver marks an array as "dirty" before writing any data to it, and marks it as "clean" when the array is being disabled, e.g. at shutdown. If the md driver finds an array to be dirty at startup, it proceeds to correct any possibly inconsistency. For RAID1, this involves copying the contents of the first drive onto all other drives. For RAID4, RAID5 and RAID6 this involves recalculating the parity for each stripe and making sure that the parity block has the correct data. For RAID10 it involves copying one of the replicas of each block onto all the others. This process, known as "resynchronising" or "resync" is performed in the background. The array can still be used, though possibly with reduced performance.

If a RAID4, RAID5 or RAID6 array is degraded (missing at least one drive, two for RAID6) when it is restarted after an unclean shutdown, it cannot recalculate parity, and so it is possible that data might be undetectably corrupted. The 2.4 md driver **does not** alert the operator to this condition. The 2.6 md driver will fail to start an array in this condition without manual intervention, though this behaviour can be overridden by a kernel parameter.

RECOVERY

If the md driver detects a write error on a device in a RAID1, RAID4, RAID5, RAID6, or RAID10 array, it immediately disables that device (marking it as faulty) and continues operation on the remaining devices. If there are spare drives, the driver will start recreating on one of the spare drives the data which was on that failed drive, either by copying a working drive in a RAID1 configuration, or by doing calculations with the

parity block on RAID4, RAID5 or RAID6, or by finding and copying originals for RAID10.

In kernels prior to about 2.6.15, a read error would cause the same effect as a write error. In later kernels, a read-error will instead cause md to attempt a recovery by overwriting the bad block. i.e. it will find the correct data from elsewhere, write it over the block that failed, and then try to read it back again. If either the write or the re-read fail, md will treat the error the same way that a write error is treated, and will fail the whole device.

While this recovery process is happening, the md driver will monitor accesses to the array and will slow down the rate of recovery if other activity is happening, so that normal access to the array will not be unduly affected. When no other activity is happening, the recovery process proceeds at full speed. The actual speed targets for the two different situations can be controlled by the **speed_limit_min** and **speed_limit_max** control files mentioned below.

SCRUBBING AND MISMATCHES

As storage devices can develop bad blocks at any time it is valuable to regularly read all blocks on all devices in an array so as to catch such bad blocks early. This process is called*scrubbing*.

md arrays can be scrubbed by writing either *check* or *repair* to the file *md/sync_action* in the *sysfs* directory for the device.

Requesting a scrub will cause *md* to read every block on every device in the array, and check that the data is consistent. For RAID1 and RAID10, this means checking that the copies are identical. For RAID4, RAID5, RAID6 this means checking that the parity block is (or blocks are) correct.

If a read error is detected during this process, the normal read-error handling causes correct data to be found from other devices and to be written back to the faulty device. In many case this will effectively *fix* the bad block.

If all blocks read successfully but are found to not be consistent, then this is regarded as a *mismatch*.

If *check* was used, then no action is taken to handle the mismatch, it is simply recorded. If *repair* was used, then a mismatch will be repaired in the same way that *resync* repairs arrays. For RAID5/RAID6 new parity blocks are written. For RAID1/RAID10, all but one block are overwritten with the content of that one block.

A count of mismatches is recorded in the *sysfs* file *md/mismatch_cnt*. This is set to zero when a scrub starts and is incremented whenever a sector is found that is a mismatch. *md* normally works in units much larger than a single sector and when it finds a mismatch, it does not determine exactly how many actual sectors were affected but simply adds the number of sectors in the IO unit that was used. So a value of 128 could simply mean that a single 64KB check found an error (128 x 512bytes = 64KB).

If an array is created by *mdadm* with *−−assume−clean* then a subsequent check could be expected to find some mismatches.

On a truly clean RAID5 or RAID6 array, any mismatches should indicate a hardware problem at some level - software issues should never cause such a mismatch.

However on RAID1 and RAID10 it is possible for software issues to cause a mismatch to be reported. This does not necessarily mean that the data on the array is corrupted. It could simply be that the system does not care what is stored on that part of the array - it is unused space.

The most likely cause for an unexpected mismatch on RAID1 or RAID10 occurs if a swap partition or swap file is stored on the array.

When the swap subsystem wants to write a page of memory out, it flags the page as 'clean' in the memory manager and requests the swap device to write it out. It is quite possible that the memory will be changed while the write-out is happening. In that case the 'clean' flag will be found to be clear when the write completes and so the swap subsystem will simply forget that the swapout had been attempted, and will possibly choose a different page to write out.

If the swap device was on RAID1 (or RAID10), then the data is sent from memory to a device twice (or more depending on the number of devices in the array). Thus it is possible that the memory gets changed between the times it is sent, so different data can be written to the different devices in the array. This will be detected by *check* as a mismatch. However it does not reflect any corruption as the block where this mismatch occurs is being treated by the swap system as being empty, and the data will never be read from that block.

It is conceivable for a similar situation to occur on non-swap files, though it is less likely.

Thus the *mismatch_cnt* value can not be interpreted very reliably on RAID1 or RAID10, especially when the device is used for swap.

BITMAP WRITE-INTENT LOGGING

From Linux 2.6.13, *md* supports a bitmap based write-intent log. If configured, the bitmap is used to record which blocks of the array may be out of sync. Before any write request is honoured, md will make sure that the corresponding bit in the log is set. After a period of time with no writes to an area of the array, the corresponding bit will be cleared.

This bitmap is used for two optimisations.

Firstly, after an unclean shutdown, the resync process will consult the bitmap and only resync those blocks that correspond to bits in the bitmap that are set. This can dramatically reduce resync time.

Secondly, when a drive fails and is removed from the array, md stops clearing bits in the intent log. If that same drive is re-added to the array, md will notice and will only recover the sections of the drive that are covered by bits in the intent log that are set. This can allow a device to be temporarily removed and reinserted without causing an enormous recovery cost.

The intent log can be stored in a file on a separate device, or it can be stored near the superblocks of an array which has superblocks.

It is possible to add an intent log to an active array, or remove an intent log if one is present.

In 2.6.13, intent bitmaps are only supported with RAID1. Other levels with redundancy are supported from 2.6.15.

BAD BLOCK LIST

From Linux 3.5 each device in an *md* array can store a list of known-bad-blocks. This list is 4K in size and usually positioned at the end of the space between the superblock and the data.

When a block cannot be read and cannot be repaired by writing data recovered from other devices, the address of the block is stored in the bad block list. Similarly if an attempt to write a block fails, the address will be recorded as a bad block. If attempting to record the bad block fails, the whole device will be marked faulty.

Attempting to read from a known bad block will cause a read error. Attempting to write to a known bad block will be ignored if any write errors have been reported by the device. If there have been no write

errors then the data will be written to the known bad block and if that succeeds, the address will be removed from the list.

This allows an array to fail more gracefully - a few blocks on different devices can be faulty without taking the whole array out of action.

The list is particularly useful when recovering to a spare. If a few blocks cannot be read from the other devices, the bulk of the recovery can complete and those few bad blocks will be recorded in the bad block list.

RAID WRITE HOLE

Due to non-atomicity nature of RAID write operations, interruption of write operations (system crash, etc.) to RAID456 array can lead to inconsistent parity and data loss (so called RAID-5 write hole). To plug the write hole md supports two mechanisms described below.

DIRTY STRIPE JOURNAL

From Linux 4.4, md supports write ahead journal for RAID456. When the array is created, an additional journal device can be added to the array through write-journal option. The RAID write journal works similar to file system journals. Before writing to the data disks, md persists data AND parity of the stripe to the journal device. After crashes, md searches the journal device for incomplete write operations, and replay them to the data disks.

When the journal device fails, the RAID array is forced to run in read-only mode.

PARTIAL PARITY LOG

From Linux 4.12 md supports Partial Parity Log (PPL) for RAID5 arrays only. Partial parity for a write operation is the XOR of stripe data chunks not modified by the write. PPL is stored in the metadata region of RAID member drives, no additional journal drive is needed. After crashes, if one of the not modified data disks of the stripe is missing, this updated parity can be used to recover its data.

This mechanism is documented more fully in the file Documentation/md/raid5-ppl.rst

WRITE-BEHIND

From Linux 2.6.14, *md* supports WRITE-BEHIND on RAID1 arrays.

This allows certain devices in the array to be flagged as *write-mostly*. MD will only read from such devices if there is no other option.

If a write-intent bitmap is also provided, write requests to write-mostly devices will be treated as write-behind requests and md will not wait for writes to those requests to complete before reporting the write as complete to the filesystem.

This allows for a RAID1 with WRITE-BEHIND to be used to mirror data over a slow link to a remote computer (providing the link isn't too slow). The extra latency of the remote link will not slow down normal operations, but the remote system will still have a reasonably up-to-date copy of all data.

FAILFAST

From Linux 4.10, *md* supports FAILFAST for RAID1 and RAID10 arrays. This is a flag that can be set on individual drives, though it is usually set on all drives, or no drives.

When *md* sends an I/O request to a drive that is marked as FAILFAST, and when the array could survive the loss of that drive without losing data, *md* will request that the underlying device does not perform any

retries. This means that a failure will be reported to *md* promptly, and it can mark the device as faulty and continue using the other device(s). *md* cannot control the timeout that the underlying devices use to determine failure. Any changes desired to that timeout must be set explicitly on the underlying device, separately from using *mdadm*.

If a FAILFAST request does fail, and if it is still safe to mark the device as faulty without data loss, that will be done and the array will continue functioning on a reduced number of devices. If it is not possible to safely mark the device as faulty, *md* will retry the request without disabling retries in the underlying device. In any case, *md* will not attempt to repair read errors on a device marked as FAILFAST by writing out the correct. It will just mark the device as faulty.

FAILFAST is appropriate for storage arrays that have a low probability of true failure, but will sometimes introduce unacceptable delays to I/O requests while performing internal maintenance. The value of setting FAILFAST involves a trade-off. The gain is that the chance of unacceptable delays is substantially reduced. The cost is that the unlikely event of data-loss on one device is slightly more likely to result in data-loss for the array.

When a device in an array using FAILFAST is marked as faulty, it will usually become usable again in a short while. *mdadm* mak es no attempt to detect that possibility. Some separate mechanism, tuned to the specific details of the expected failure modes, needs to be created to monitor devices to see when they return to full functionality, and to then re-add them to the array. In order of this "re-add" functionality to be effective, an array using FAILFAST should always have a write-intent bitmap.

RESTRIPING

Restriping, also known as *Reshaping*, is the processes of re-arranging the data stored in each stripe into a new layout. This might involve changing the number of devices in the array (so the stripes are wider), changing the chunk size (so stripes are deeper or shallower), or changing the arrangement of data and parity (possibly changing the RAID level, e.g. 1 to 5 or 5 to 6).

As of Linux 2.6.35, md can reshape a RAID4, RAID5, or RAID6 array to have a different number of devices (more or fewer) and to have a different layout or chunk size. It can also convert between these different RAID levels. It can also convert between RAID0 and RAID10, and between RAID0 and RAID4 or RAID5. Other possibilities may follow in future kernels.

During any stripe process there is a 'critical section' during which live data is being overwritten on disk. For the operation of increasing the number of drives in a RAID5, this critical section covers the first few stripes (the number being the product of the old and new number of devices). After this critical section is passed, data is only written to areas of the array which no longer hold live data — the live data has already been located away.

For a reshape which reduces the number of devices, the 'critical section' is at the end of the reshape process.

md is not able to ensure data preservation if there is a crash (e.g. power failure) during the critical section. If md is asked to start an array which failed during a critical section of restriping, it will fail to start the array.

To deal with this possibility, a user-space program must

- Disable writes to that section of the array (using the **sysfs** interface),
- take a copy of the data somewhere (i.e. make a backup),
- allow the process to continue and invalidate the backup and restore write access once the critical section is passed, and

- provide for restoring the critical data before restarting the array after a system crash.

mdadm versions from 2.4 do this for growing a RAID5 array.

For operations that do not change the size of the array, like simply increasing chunk size, or converting RAID5 to RAID6 with one extra device, the entire process is the critical section. In this case, the restripe will need to progress in stages, as a section is suspended, backed up, restriped, and released.

SYSFS INTERFACE

Each block device appears as a directory in *sysfs* (which is usually mounted at **/sys**). For MD devices, this directory will contain a subdirectory called **md** which contains various files for providing access to information about the array.

This interface is documented more fully in the file **Documentation/admin-guide/md.rst** which is distributed with the kernel sources. That file should be consulted for full documentation. The following are just a selection of attribute files that are available.

md/sync_speed_min

This value, if set, overrides the system-wide setting in **/proc/sys/dev/raid/speed_limit_min** for this array only. Writing the value **system** to this file will cause the system-wide setting to have effect.

md/sync_speed_max

This is the partner of **md/sync_speed_min** and overrides **/proc/sys/dev/raid/speed_limit_max** described below.

md/sync_action

This can be used to monitor and control the resync/recovery process of MD. In particular, writing "check" here will cause the array to read all data block and check that they are consistent (e.g. parity is correct, or all mirror replicas are the same). Any discrepancies found are **NOT** corrected.

A count of problems found will be stored in **md/mismatch_count**.

Alternately, "repair" can be written which will cause the same check to be performed, but any errors will be corrected.

Finally, "idle" can be written to stop the check/repair process.

md/stripe_cache_size

This is only available on RAID5 and RAID6. It records the size (in pages per device) of the stripe cache which is used for synchronising all write operations to the array and all read operations if the array is degraded. The default is 256. Valid values are 17 to 32768. Increasing this number can increase performance in some situations, at some cost in system memory. Note, setting this value too high can result in an "out of memory" condition for the system.

memory_consumed = system_page_size * nr_disks * stripe_cache_size

md/preread_bypass_threshold

This is only available on RAID5 and RAID6. This variable sets the number of times MD will service a full-stripe-write before servicing a stripe that requires some "prereading". For fairness this defaults to 1. Valid values are 0 to stripe_cache_size. Setting this to 0 maximizes sequential-write throughput at the cost of fairness to threads doing small or random writes.

md/bitmap/backlog
The value stored in the file only has any effect on RAID1 when write-mostly devices are active, and write requests to those devices are proceed in the background.

This variable sets a limit on the number of concurrent background writes, the valid values are 0 to 16383, 0 means that write-behind is not allowed, while any other number means it can happen. If there are more write requests than the number, new writes will by synchronous.

md/bitmap/can_clear
This is for externally managed bitmaps, where the kernel writes the bitmap itself, but metadata describing the bitmap is managed by mdmon or similar.

When the array is degraded, bits mustn't be cleared. When the array becomes optimal again, bit can be cleared, but first the metadata needs to record the current event count. So md sets this to 'false' and notifies mdmon, then mdmon updates the metadata and writes 'true'.

There is no code in mdmon to actually do this, so maybe it doesn't even work.

md/bitmap/chunksize
The bitmap chunksize can only be changed when no bitmap is active, and the value should be power of 2 and at least 512.

md/bitmap/location
This indicates where the write-intent bitmap for the array is stored. It can be "none" or "file" or a signed offset from the array metadata - measured in sectors. You cannot set a file by writing here - that can only be done with the SET_BITMAP_FILE ioctl.

Write 'none' to 'bitmap/location' will clear bitmap, and the previous location value must be write to it to restore bitmap.

md/bitmap/max_backlog_used
This keeps track of the maximum number of concurrent write-behind requests for an md array, writing any value to this file will clear it.

md/bitmap/metadata
This can be 'internal' or 'clustered' or 'external'. 'internal' is set by default, which means the metadata for bitmap is stored in the first 256 bytes of the bitmap space. 'clustered' means separate bitmap metadata are used for each cluster node. 'external' means that bitmap metadata is managed externally to the kernel.

md/bitmap/space
This shows the space (in sectors) which is available at md/bitmap/location, and allows the kernel to know when it is safe to resize the bitmap to match a resized array. It should big enough to contain the total bytes in the bitmap.

For 1.0 metadata, assume we can use up to the superblock if before, else to 4K beyond superblock. For other metadata versions, assume no change is possible.

md/bitmap/time_base
This shows the time (in seconds) between disk flushes, and is used to looking for bits in the bitmap to be cleared.

The default value is 5 seconds, and it should be an unsigned long value.

KERNEL PARAMETERS

The md driver recognised several different kernel parameters.

raid=noautodetect

This will disable the normal detection of md arrays that happens at boot time. If a drive is partitioned with MS-DOS style partitions, then if any of the 4 main partitions has a partition type of 0xFD, then that partition will normally be inspected to see if it is part of an MD array, and if any full arrays are found, they are started. This kernel parameter disables this behaviour.

raid=partitionable

raid=part

These are available in 2.6 and later kernels only. They indicate that autodetected MD arrays should be created as partitionable arrays, with a different major device number to the original non-partitionable md arrays. The device number is listed as *mdp* in */proc/devices*.

md_mod.start_ro=1

/sys/module/md_mod/parameters/start_ro

This tells md to start all arrays in read-only mode. This is a soft read-only that will automatically switch to read-write on the first write request. However until that write request, nothing is written to any device by md, and in particular, no resync or recovery operation is started.

md_mod.start_dirty_degraded=1

/sys/module/md_mod/parameters/start_dirty_degraded

As mentioned above, md will not normally start a RAID4, RAID5, or RAID6 that is both dirty and degraded as this situation can imply hidden data loss. This can be awkward if the root filesystem is affected. Using this module parameter allows such arrays to be started at boot time. It should be understood that there is a real (though small) risk of data corruption in this situation.

md=*n***,***dev***,***dev***,...**

md=d*n***,***dev***,***dev***,...**

This tells the md driver to assemble **/dev/md n** from the listed devices. It is only necessary to start the device holding the root filesystem this way. Other arrays are best started once the system is booted.

In 2.6 kernels, the **d** immediately after the = indicates that a partitionable device (e.g. **/dev/md/d0**) should be created rather than the original non-partitionable device.

md=*n***,***l***,***c***,***i***,***dev...*

This tells the md driver to assemble a legacy RAID0 or LINEAR array without a superblock. *n* gives the md device number, *l* gives the level, 0 for RAID0 or −1 for LINEAR, *c* gives the chunk size as a base-2 logarithm offset by twelve, so 0 means 4K, 1 means 8K. *i* is ignored (le gacy support).

FILES

/proc/mdstat

Contains information about the status of currently running array.

/proc/sys/dev/raid/speed_limit_min
A readable and writable file that reflects the current "goal" rebuild speed for times when non-rebuild activity is current on an array. The speed is in Kibibytes per second, and is a per-device rate, not a per-array rate (which means that an array with more disks will shuffle more data for a given speed). The default is 1000.

/proc/sys/dev/raid/speed_limit_max
A readable and writable file that reflects the current "goal" rebuild speed for times when no non-rebuild activity is current on an array. The default is 200,000.

SEE ALSO

mdadm(8),