md

Pilote Multiple Device

Le pilote md fournis des périphériques virtuels qui sont créés depuis un ou plusieurs périphériques indépendants. Cet array de périphériques contient souvent une redondance et les périphériques sont souvent des périphériques disques.

md supporte les RAID niveau 1 (mirroring), 4 (striped avec parité), 5 (striped avec parité distribué), 6 (striped avec double redondance), et 10 (striped et mirroré). Si un périphérique échoue en utilisant un de ces levels, l'array contitue de fonctionner.

md supporte également les configurations pseudo-raid (non-redondants) incluant le raid0 (striped array), linear (array sérialisé), multipath (un jeu de différentes interfaces vers le même périphérique) et faulty (une couche sur un simple périphérique dans lequel les erreurs sont injectés).

Métadonnées MD

Chaque périphérique dans un array peut avoir des métadonnées stockées dans le périphérique. Ces métadonnées sont parfois appelés superblock. Les métadonnées enregistrent des informations sur la structure et l'état de l'array pour pouvoir ré-assembler l'array après un arrêt.

Depuis la version 2.6.10 du Kernel, md fournis le support pour 2 formats différents de métadonnées, et d'autres formats peuvent être ajoutés.

Le format commun, la version 0.90, a un superblock qui fait 4ko et est écrit dans un block aligné sur 64Ko qui démarre au plus 64K et moins de 128k de la fin du périphérique (ex, pour obtenir l'adresse du superblock arrondir la taille du périphérique à un multiple de 64K, puis soustraire 64K). La taille disponible pour chaque périphérique est la quantité d'espace avent le superblock, dont entre 64K et 128K est perdu quant un périphérique est incorporé dans un array MD. Ce superblock stock les champs multi-octet de manière indépendante su processeur, pour que les array puissent être déplacés entre des machines de différents processeurs.

Le nouveau format, version 1, a un superblock qui fait normalement 1K de long, mais peut faire plus. Il est normalement stocké entre 8K et 12k de la fin du périphérique, sur un alignement 4K, bien de des variante peuvent être stockées au début du disque (version 1.1) ou à 4K du début du périphérique (version 1.2). Ce format de métadonnées stocke les données multi-octets dans un format indépendant du processeur et supporte des centaines de périphériques contre 28 pour les versions 0.90.

Les métadonnées contiennent, entre autre :

LEVEL Le type d'array

UUID Un identifiant 128bits unique qui identifie l'array

Quand une versions 0.90 est redéfinie (ex, ajout de péripériques dans un raid 5), le numéro de version est temporairement mis à 0.91. Celà permet de s'assurer que le processus s'est terminé correctement.

Métadonnées sans arrays

Bien qu'il est recommandé de créer des array avec des superblocks pour qu'ils puissent être réassemblés correctement, il y a certaines circonstances où un array sant superblock est préférrable :

LEGACY ARRAY Les première versions du pilote md ne supportent que les configuration LINEAR et RAID0 et n'utilise pas de superblock (qui est moins critique dans ces configuration).

FAULTY Ce type ne tire aucun bénéfice d'un superblock

MULTIPATH Il est souvent possible de détecter les périphériques qui sont des chemins différents du même stockage directement au lieu d'avoir un superblock distinct. Dans ce cas, un array multipath sans superblock a du sens.

RAID1 Dans certaines configurations il peut être souhaitable de créer une configuration raid1 sans superblock, pour maintenir l'état de l'array n'importe où.

Métadonnées externes

Depuis le kernel 2.6.28, le pilote md supporte les arrays avec des métadonnées externe. C'est à dire que les métadonnées ne sont pas gérés par le kernel mais par un programme de l'espace utilisateur. Cela permet de supporter différents formats de métadonnées.

md est capable de communiquer avec le programme externe via différents attributs sysfs pour qu'il puisse effectuer les changements appropriés - par exemple, pour marquer un périphérique comme étant en faute. Si nécessaire, md attend que le programme prenne connaissance de l'événement en écrivant dans un attribut sysfs. la man mdmon(8) contient des détails sur cette intéraction.

Conteneurs

LINEAR

Un array LINEAR sérialise simplement l'espace disponible de chaque lecteur pour former un grand disque virtuel. L'avantage de cet arrangement par rapport au raid0 est que l'array peut être configuré plus tard sans perturber les données dans l'array.

RAID0

Un raid0 est également connu comme array striped. Il est configuré à la création avec un taille de shunk qui doit être une puissance de 2, et au moins 4 KiO. Le premier chunk est assigné au premier disque, le second au deuxième disque, etc. Cette collection de chunk forment un stripe. Si les périphériques dans l'array n'ont pas tous la même taille, une fois le plus petit disque remplis, le pilote collecte les chunks en stripes plus petits.

RAID1

Un raid1 est également connus comme un jeu mirroir. Une fois initialisé, chaque périphérique dans un raid1 devrait avoir la même taille. Si ce n'est pas le cas, seul la quantité d'espace disponible sur le plus petit périphérique est utilisé.

Noter que la lecture balancé faite par le pilote n'améliore pas les performance comme un raid0. Les périphériques individuels dans un raid1 peuvent être marqués 'write-mostly'. Ces lecteurs sont exclus de la lecture balancée et seront lus seulement s'il n'y a pas d'autre option. Peut être utile pour des périphériques connectés sur un lien lent.

RAID4

Un raid4 est similaire à un RAID0 avec un périphérique suplémentaire pour stocker la parité. Ce périphérique est le dernier des périphériques actifs dans l'array. à la différence de raid0, raid4 nécessite également que tous les stripes disques, donc l'espace

supplémentaire sur les disques plus grand que le plus petit est perdu.

Cela permet à l'array de continuer de fonctionner si un disque échoue.

RAID5

Similaire au raid5, mais les blocks de parités pour chaque stripe est distribué sur tous les périphériques. Cela permet plus de parallelisme dans les écritures. Il y a également plus de parallélisme pendant les lecture.

RAID6

Similaire au raid5, mais peut gérer la perte de 2 périphériques sans perte de données. Nécessite N+2 disques. Les performances d'un raid6 est légerement inférieur à un raid5 en mode simple disque d'erreur, et très lent pour un mode double disque d'erreur.

RAID10

Fournis une combinaison de raid1 et raid0, et parfois appelé raid1+0. Chaque block de données est dupliqué un certain nombre de fois, et la collection résultante de ces blocks sont distribués sur plusieurs disques.

En configurant un raid10, il est nécessaire de spécifier le nombre de réplicas de chaque block de données qui sont requis (généralement 2) et si leur layout est 'near', 'far' ou 'offset'

Exemples de layout raid10

Les exemples ci-dessous visualisent la distribution de chunk dans les périphériques sous-jacents pour le layout.

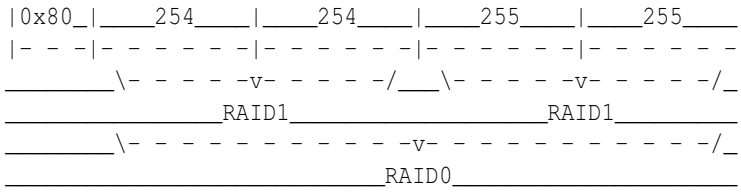
Pour simplifier il est assumé que la taille des chunks est égal à la taille des blocks des périphériques sous-jacents tout comme ceux du périphérique RAID10 exporté par le kernel.

Les nombres décimaux (0, 1, 2, ...) sont les chunks du RAID10 et donc également des blocks et adresse de blocks du périphérique raid exporté

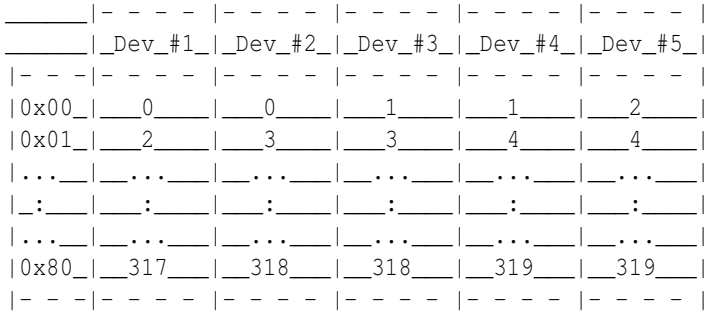
Layout "near" Quand des réplicas 'near' sont choisis, les copies multiples d'un chunk donné sont disposés consécutivement (aussi proche les uns des autres que possible) dans les stripes de l'array.

Avec un nombre paire de périphériques, ils sont probablement être au même offset sur les différents périphériques. C'est le raid1+0 classique ; c'est à dire 2 groupes de périphériques mirrorés. Dans l'exemple ci-dessous, les groupes de périphériques #1 / #2 et #3 / #4 sont des raid1, formant un raid0.

```
______|- - - - - -|- - - - - -|- - - - - -|- - - - - -|
______|_Device_#1_|_Device_#2_|_Device_#3_|_Device_#4_|
|- - -|- - - - - -|- - - - - -|- - - - - -|- - - - - -|
|0x00_|_____0_____|_____0_____|_____1_____|_____1_____|
|0x01_|_____2_____|_____2_____|_____3_____|_____3_____|
|..._|____..._____|____..._____|____..._____|____..._____|
|_:___|_____:_____|_____:_____|_____:_____|_____:_____|
|..._|____..._____|____..._____|____..._____|____..._____|
```



Exemple avec 2 copies par chunk et un nombre impaire de périphériques :



Layout "far" Quand les réplicas far sont choisis, les copies d'un chunk sont disposés assez éloignés (aussi loin qu'il est raisonnablement possible)

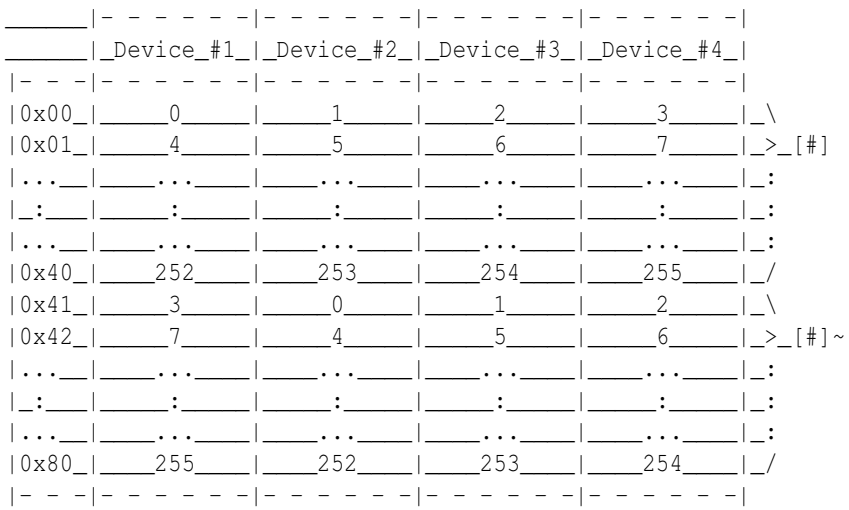
D'abord une séquence complète de tous les blocks de données (c'est à dire la données vues sur le périphériques blocks raid10 exporté) sont stripé sur les périphériques Puis une autre séquence (shift) complète de tous les blocks de données ; et ainsi de suite

Le shift nécessaire pour empêcher de placer les copies du même chunks sur les même périphériques est actuellement une permutation cyclique avec l'offset 1 de chaque stripes dans une séquence complet de chunks

L'offset 1 est relatif à la séquence complète précédente de chunks, donc en cas de plus de 2 copies par chunk on obtient les offsets suivantes :

```
1. Séquence complète de chunks: offset = 0
2. Séquence complète de chunks: offset = 1
3. Séquence complète de chunks: offset = 2
...
n. complete sequence of chunks: offset = n-1
```

Exemple avec 2 copies par chunk et un nombre paire de périphériques :



Exemple avec 2 copies par chunk et un nombre impaire de périphériques :

```
_______|- - - - - |- - - - - |- - - - - |- - - - - |- - - - - |
_______|_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____|_/
|- - -|- - - - - |- - - - - |- - - - - |- - - - - |- - - - - |
```

Avec [#] étant la séquence complète de chunks et [#]~la permutation cyclique avec l'offset 1 (dans le cas de plus de 2 copies par chunk ce serait ([#]~)~, (([#]~)~)~, ...)

L'avantage de ce layout est que MD peut facilement propager les lectures séquentiellement sur le périphériques, les rendant similaire au raid0 en terme de vitesse. Les écriture en revenche sont plus lents

Layout offset Quand des réplicas offset sont choisis, toutes les copies d'un chunk donné sont stripé consécutivement (offset par la longueur du stripe avec chacun d'entre eux) sur les périphériques.

Expliqué en détail, les chunks consécutif sont stripés sur les périphériques, immédiatement suivis par une copie décalée de ces chunk. Ce motif se répète pour tous les chunks consécutifs suivant.

Le shift est nécessaire pour empêcher de placer les copies des même chunks sur le même périphérique avec une permutation de 1 de chaque copie stripée de chunks consécution.

L'offset 1 est relatif à la copie précedente, donc dans le cas de plus de 2 copies par chunk on obtient :

```
1. chunks consécutifs <nombre de périphériques>: offset = 0
2. chunks consécutifs <nombre de périphériques> = 1
3. chunks consécutifs <nombre de périphériques> = 2
...
n. chunks consécutifs <nombre de périphériques> = n-1
```

Exemple avec 2 copies par chunk et un nombre paire de périphériques :

```
_______|- - - - - - -|- - - - - - -|- - - - - - -|- - - - - - -|
_______|_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~
|- - -|- - - - - - -|- - - - - - -|- - - - - - -|- - - - - - -|
```

Exemple avec 2 copies par chunk et un nombre impaire de périphériques :

```
_______|- - - - - |- - - - - |- - - - - |- - - - - |- - - - - |
```

```
_______|_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~
|- - -|- - - - |- - - - |- - - - |- - - - |- - - - |
```

Avec AA, AB, ..., AZ, BA, ... étant le jeu de <nombre de périphériques> consécutifs de chunks et AA~, AB~, ..., AZ~, BA~, ... la permutation cyclique avec l'offset 1 (dans le cas de plus de 2 copies par chunk, il y aura (AA~)~, ... et ((AA~)~)~, ... et ainsi de suite)

Cela donne des caractéristiques de lecture similaires à far, mais plus adapté à de grandes tailles de chunk, sans autant d'écritures.

Il devrait être noté que le nombre de périphériques dans un raid10 n'a pas besoin d'être un multiple du nombre de réplica de chaque block de données ; cependant, il doit y avoir au moins autant de périphériques que de réplicas

Si, par exemple, un array est créé avec 5 périphériques et 2 réplicas, alors un espace équivalent à 2,5 périphériques sera disponible, et chaque block sera stocké sur 2 périphériques différents.

Finalement, il est possible d'avoir un array avec des copies à la fois near et far. Si un array est configuré avec 2 copies near et 2 copies far, il y aura un total de 4 copies de chaque block, chacun sur un disque différent.

MULTIPATH MULTIPATH n'est pas vraiment un RAID vu qu'il n'y a qu'un disque physique dans un array md MULTIPATH. Cependant il y a plusieurs points d'accès à ce périphérique, et un de ces chemins peut échouer, donc il y a quelques similarités.

Un array MULTIPATH est composé d'un nombre de périphériques logiquement différents, souvent des interfaces fibre qui réfèrent tous au même périphérique. Si une de ces interfaces échoue (ex : dûs à un problème de cable), le pilote va tenter de rediriger les requêtes vers une autre interface.

Le disque MULTIPATH ne reçoit plus de développement et ne devrait pas être utilisé.

FAULTY Le module md FAULTY est fournis dans un but de tests. Un array FAULTY a exactement un périphérique et est normalement assemblé sans superblock, dont l'array créé fournis un accès directe à toutes les données dans le périphérique.

Le module FAULTY peut être requis pour simuler des fautes pour permettre de tester d'autres niveaux md ou systèmes de fichier. Les fautes peuvent être choisies pour se déclencher sur des requêtes de lecture, et peut être transitoires ou persistantes.

Arrêt non propre

Quand des changements sont fait sur un array RAID1/4/5/6/10 il y a une possibilité d'inconsistance pour de courtes périodes de temps vu que chaque mise à jours nécessite d'écrire au moins 2 blocks sur différents périphériques, et ces écritures peuvent ne pas se produire au même moment. Donc si un système avec un de ces array est éteind au milieu d'une opération d'écriture, l'array peut être inconsistant.

Pour gérer cette situation, le pilote md marque l'array 'dirty' avant d'écrire une donnée, et le marque 'clean' quan l'array est désactivé à l'extinction. Si le pilote md trouve un array dirty au démarrage, il procède à une correction. Pour RAID1, cela implique de copier le contenu du premier disque dans tous les autres. Pour RAID4/5/6, cela implique de recalculer la parité pour chaque stripe et de s'assurerque le block de parité a une valeur correcte. Pour RAID10 cela implique de copier un des réplicas de chaque block dans tous les autres. Ce processus de resynchronisation est effectue en tâche de fond. L'array peut être utilisé pendant ce temps, mais possiblement avec des performances réduites.

Si un RAID4/5/6 est dégradé (il manque au moins un disque, 2 pour un raid6) quand il est redémarré après un arrêt non propre, il ne peut pas recalculer la parité, et des données corrompues peuvent ne pas être détectés. Le pilote md ne démarrera pas si un array dans ces condition sans intervention manuelle, bien que ce comportement peut être changé par un paramètre kernel.

Récupération

Si le pilote md détecte une erreur d'écriture sur un périphérique dans un RAID1/4/5/6/10, il désactive immédiatement le périphérique (marqué faulty) et continue ses opérations sur les autres disques. Il y a un disque spare, le pilote va démarrer le re-création sur un des disques spare.

Dans le kernel, une erreur de lecture force md à ne pas tenter de récupérer un mauvais block, ex, il va trouve la donnée correcte quelque-part, écrire sur le block défectueux, puis tenter de lire ce block de nouveau. Si l'écriture ou la re-lecture échoue, md traite l'erreur de la même manière qu'une erreur d'écriture est traitée, et échoue tout le périphérique.

Quand ce processus de récupération se produit, md monitor l'accès à l'array et affiche le taux de récupération si d'autre activités se produisent, donc un accès normal à l'array n'est pas affecté. Quand une autre activité survient, le processus de récupération la traite à pleine vitesse. Les vitesses sont contrôlés avec 'speed_limit_min' et 'speed_limit_max'.

scrubbing et mismatches

Vu que les périphériques de stockage peuvent développer de mauvais blocks à tout moment il est utile de lire régulièrement tous les blocks de tous les périphériques dans un array pour capturer ces blocks defectueux le plus tôt possible. Ce processus est appelé scribbing.

Les arrays md peuvent être scrubbés en écrivant soit check ou repair dans le fichier md/sync_action dans le répertoire sysfs pour le périphérique.

En demandant un scrub, md lit tous les block et vérifie la consistance des données. Pour raid1/10, cela signifie que les copies sont identiques. pour raid4/5/6 cela signifie que le block de parité est correcte.

Si une erreur de lecture est détectée durant ce processus, le gestion d'erreur corrige les données. Dans de nombreux cas, cela corrige effectivement le mauvais block.

Si tous les blocks sont lus avec succès mais apparaîssent inconsistants, ils sont considérés comme mismatch.

Si 'check' a été utilisée, aucune action n'est prise pour gérer le mismatch, il est simplement enregistré. Si 'repair' est utilisé, un mismatch sera réparé de la même manière que resync répare les array. Pour raid5/6, de nouveaux blocks de parité sont écris. Pour raid1/10, tous sauf un block sont écrasés avec le contenu de ce block.

Un compteur de mismatch est enregistré dans sysfs 'md/mismatch_cnt'. Il est mis à 0 quand un scrub commence et est incrémenté quand un secteur est trouvé qui est un mismatch, il ne détermine pas exactement combien de secteurs sont affectés mais ajoute simplement le nombre de secteurs dans l'unité IO qui été utilisé. Donc une valeur de 128 peut signifier qu'une vérification de 64Ko a trouvé une erreur (128*512o = 64Ko)

Si un array est créé par mdadm avec –assume-clean alor une vérification ultérieure peut s'attendre à trouver des mismatch. Dans un raid5/6 propre, tout mismatch devrait indiquer un problème hardware.

Cependant, sur un raid1/10 il est possible qu'il s'agisse d'un problème logiciel. Cela ne signifie pas nécessairement que les données dans l'array sont corrompues. Cela peut simplement signifier que le système ne s'occupe pas de ce qui est stocké sur cette partie de l'array - c'est de l'espace inutilisé.

La cause principale de mismatch sur un raid0/10 se produit si une partition swap ou un fichier swap est stockés dans l'array.

Quand le sous-système swap souhaite écrire une page de mémoire, il flag la page comme 'clean' dans le gestionnaire mémoire et demande au périphériques swap de l'écrire. Il est possible que la mémoire soit changée durant l'écriture dans le swap. Dans ce cas le flag 'clean' sera effacé une fois l'écriture complétée, et le sous-système swap va simplement oublier que le swapout a été tenté, et va possiblement choisir une page différente à écrire.

Si le périphériques swap était sur un raid1/10, la donnée est envoyée de la mémoire vers de périphérique 2 fois, donc il est possible que la mémoire ait été changée entre temps. Ainsi, la valeur mismatch_cnt ne peut pas être interprété de manière sûre sur un raid1 ou raid10, notamment quand le périphérique est utilisé pour le swap.

Bitmap write-intent logging

md supporte le log d'écriture basé sur bitmap. Si configuré, le bitmap est utilisé pour enregistrer quels blocks de l'array peuvent être désyncho. Avant toute requête d'écriture, md s'assure que le bit correspondant dans le log est mis. Après un délai sant écritures dans une zone de l'array, le bit correspondant est effacé.

Ce bitmap est utilisé pour 2 optimisations :

D'abord, après un arrêt non propre, le processus de resyncho va consulter le bitmap et seulement resynchroniser les blocks qui correspondent aux bits mis dans le bitmap. Cela peut considérablement réduire le temps de resynchro

Ensuite, quand un lecteur échoue et est enlevé de l'array, md arrête d'effacer les bits dans les log. Si ce même disque est réajouté à l'array, md va le notifier et seulement récupérer les section du disque qui sont couverts par les bits mis dans le log. Cela peut permettre à un périphérique d'être temporairement enlevé puis réinséré sans causer de gros coûts de récupération.

Le log d'écriture peut être stocké dans un fichier sur un périphérique séparé, ou peut être stocké près des superblocks d'un array qui a des superblocks. Il est possible d'ajouter un log à un array actif, ou de le supprimer.

Bad block list

Chaque périphérique dans un array md peut stocker une liste de blocks défectueux connus. Cette liste a une taille de 4k et est généralement positionnée à la fin de l'espace entre le superblock et les données

Quand un block ne peut pas être lu et ne peut pas être réparé en écrivant les données récupérées depuis les autres périphériques, l'adresse du block est stockée dans la liste des blocks défectueux. Similairement si une tentative d'écriture dans un block échoue, l'adresse sera enregistrée comme block défectueux. Si une tentative d'enregistrer le block défectueux échoue, tout le périphérique est marqué en erreur.

Tenter de lire depuis un block défectueux génère une erreur de lecture. Tenter d'écrire dans un block défectueux connus est ignoré si des erreurs d'écriture ont été reportés par le périphérique. S'il n'y a pas d'erreur d'écriture, les données sont écrites dans le block et en cas de succès, l'adresse est supprimée de la liste.

Journal d'écriture RAID456

Dû à la nature non-atomique des opérations d'écriture raid, d'interruption des opérations d'écriture (crash système, etc) sur un raid456 peut créer une inconsistance de parité et la perte de données. (également appelé RAID-5 write hole).

md supporte l'écriture dans un journal. Quand l'array est créé, un périphérique journal additionnel peut être ajouté à l'array via l'option write-journal. Ce journal fonctionne comme les journaux des systèmes de fichier. Avant d'écrire les données sur disque, md écris les données et la parité du stripe dans le périphérique journal. Après un crash, md recherche dans le périphérique journal les opérations

d'écritures incomplètes, et les écrit sur les disques.

Quand le périphérique journal échoue, l'array est forcé en mode lecture-seule.

write-behind

md supporte write-behind sur les arrays RAID1. Cela permet à certains périphériques dans l'array d'être flaggé write-mostly. MD ne lit dans ces périphériques seulement s'il n'y a pas d'autres options. Si un bitmap write-intend est également fournis, les requêtes d'écriture vers les périphérique write-mostly sont traités comme des requêtes write-behind et md n'attend pas que les écritures soient complétées avant de reporter l'écriture complétée au système de fichier.

Cela permet à un RAID1 avec write-behind d'être utilisé comme donnée miroir sur un lien lent dans une machine distante. La latence supplémentaire de lien distant ne ralentira pas les opérations normales, mais le système distant continura à maintenir une copie raisonablement à jour de toutes les données

RESTRIPING

re restriping, également appelé reshaping, est le processus de ré-arrangement des données stockées dans chaque stripe dans une nouvelle couche. Cela peut être dû au changement du nombre de disques dans l'array (donc les stripes sont plus long), changer la taille de chunk, ou changer l'arrangement des données et de la parité (possiblement changer le niveau de raid).

md peut remodeler un raid4/5/6 pour avoir un nombre différent de disques et pour avoir un layout différent ou une taille de chunk différente. Il peut également convertir entre les différents niveaux de raid. Il peut également convertir entre raid0 et raid10, et entre raid0, raid4 ou raid5. D'autres possibilités peuvent suivre dans le future.

Durant un traitement de stripe il y a une section critique durant lequel les données sont écrasées sur le disque. Pour l'opération d'augmentation du nombre de disques dans un raid5, cette section critique couvre les premier stripes (le nombre étant le produit de l'ancien et du nouveau nombre de disques). Une fois cette section critique passée, les données sont seulement écrite dans les aires de l'array qui ne maintient pas de données

Pour un traitement qui réduit le nombre de périphériques, la section critique est à la fin du processus de reshaping.

md n'est pas capable de s'assurer de la préservation des données s'il y a un crash durant la section critique. Si md doit démarrer un array qui a échoué durant une section critique, il refuse de démarrer l'array.

Pour gérer ce cas, un programme userspace doit :

- désactiver les écritures dans cette section de l'array (en utilisant l'interface sysfs)
- Créer une copie des données
- permettre au processus de continuer et invalider l'accès en écriture au backup et restauration une fois la section critique passée
- fournir pour restorer les données critiques avant de redémarrer l'array après un crash.

mdadm le fait pour les array raid5. Pour les opérations qui ne changent pas la taille de l'array, par exemple augmenter a taille de chunk, ou convertir un raid5 en raid6 avec un périphériques supplémentaire, tout le processus est la section critique. Dans ce cas, le restripe doit progresser en étapes, vu qu'une section est suspendue, backupée, restripée, et relachée.

Interface sysfs

Chaque périphériques block apparaît dans sysfs. Pour les périphériques MD, ce répertoire va contenir un sous-répertoire 'md' qui contient divers fichiers pour fournir l'accès aux informations sur l'array.

Cette interface est documentée dans Documentation/md.txt :

md/sync_speed_min Si définis, écrase le paramètre système dans **/proc/sys/dev/raid/speed_limit_min** pour cet array uniquement. La valeur 'system' force l'utilisation du paramètre système.

md/sync_speed_max Idem, pour **/proc/sys/dev/raid/speed_limit_max**

md/sync_action Peut être utilisé pour superviser et contrôler les processus de resynchro/récupération. Écrire check déclenche la vérification de la consistance de tous les blocks de données. Un compteur de problèmes est stocké dans **md/mismatch_count**. 'repair' peut être écrit pour vérifier et corriger les erreurs.

md/stripe_cache_size Seulement disponible pour un raid5/6. Enregistre la taille (en pages par périphérique) du cache de stripe qui est utilisé pour synchoniser toutes les opérations d'écriture dans l'array et toutes les opérations de lecture si l'array est dégradé. Défaut : 256 (de 17 à 32768). Augmenter cette valeur améliore les performances dans certaines situations, au prix d'une consommation mémoire système accrue (memory_consumed = system_page_size addentry articles autoadd autofind autoprod createalpha createbeta createdb createprod findentry fullpowa generator.php genhtml genman genmd gentex html insert man md pdf regen setfor setfor2 sql temp temp-new-pc tex threads ToDo nr_disks addentry articles autoadd autofind autoprod createalpha createbeta createdb createprod findentry fullpowa generator.php genhtml genman genmd gentex html insert man md pdf regen setfor setfor2 sql temp temp-new-pc tex threads ToDo stripe_cache_size)

md/preread_bypass_threshold Seulement disponible pour un raid5/6. Cette variable définis le nombre de fois que md effectue un full-stripe-write avant de servire un stripe qui nécessite une pré-lecture. défaut : 1 (de 0 à stripe_cache_size). À 0, maximise l'écriture séquentielle au prix de l'équité pour les threads qui effectuent de petites écritures ou des écritures aléatoires.

Paramètres kernel

raid=noautodetect Désactive l'auto-detection des array au boot. Si un disque est partitionné avec des partitions de style MSDOS, si une des 4 partitions primaires a le type 0xFD, cette partition est normalement inspectée pour voir si elle fait partie d'un array. Ce paramètre désactive cette détection.

raid=partitionable

raid=part Indique que les array auto-détectés devraient être créés comme arrays partitionnables, avec un numéro majeur différent des array non-partitionnables. Le numéro de périphérique est listé comme 'mdp' dans /proc/devices'

md_mod.start_ro=1

/sys/module/md_mod/parameters/start_ro md démarre tous les array en lecture seule. C'est un RO soft qui bascule automatiquement en lecture-écriture à la première requête d'écriture.

md_mod.start_dirty_degraded=1

/sys/module/md_mod/parameters/start_dirty_degraded ne démarre pas de raid4/5/6 qui est dégradé. Ce paramètre permet de forcer le démarrage de tels array au boot. (utile pour /)

md=n,dev,dev,...

md=dn,dev,dev,... Indique au pilote md d'assembler '/dev/md n' depuis les périphériques listés. Seulement nécessaire pour démarrer le périphérique maintenant le système de fichier racine.

md=n,l,c,i,dev... Indique au pilote md d'assembler un array RAID0 ou LINEAR dans superblock. 'n' donne le nombre de périphériques, 'l' donnele niveau (0=RAID0 ou -1=LINEAR), 'c' donne la taille de chunk en logarithme base-2 par 12, donc 0 indique 4k, 1 indique 8k. 'i' est ignoré.

Fichiers

/proc/mdstat Contient les informations sur le status de l'array en cours d'exécution

/proc/sys/dev/raid/speed_limit_min Reflète la vitesse courante de reconstruction quand l'activité de non-reconstruction est en cours dans un array. La vitesse est en Kio/s, et est un taux par périphérique. Défaut : 1000.

/proc/sys/dev/raid/speed_limit_max Reflète la vitesse courante de reconstruction quand aucune activité de non-reconstruction est en cours dans un array. Défaut : 200,000.