Linux Block Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

BFQ (BUDGET FAIR QUEUEING)

BFQ is a proportional-share I/O scheduler, with some extra low-latency capabilities. In addition to cgroups support (blkio or io controllers), BFQ's main features are:

- BFQ guarantees a high system and application responsiveness, and a low latency for time-sensitive applications, such as audio or video players;
- BFQ distributes bandwidth, and not just time, among processes or groups (switching back to time distribution when needed to keep throughput high).

In its default configuration, BFQ privileges latency over throughput. So, when needed for achieving a lower latency, BFQ builds schedules that may lead to a lower throughput. If your main or only goal, for a given device, is to achieve the maximum-possible throughput at all times, then do switch off all low-latency heuristics for that device, by setting low_latency to 0. See Section 3 for details on how to configure BFQ for the desired tradeoff between latency and throughput, or on how to maximize throughput.

As every I/O scheduler, BFQ adds some overhead to per-I/O-request processing. To give an idea of this overhead, the total, single-lock-protected, per-request processing time of BFQ—i.e., the sum of the execution times of the request insertion, dispatch and completion hooks—is, e.g., 1.9 us on an Intel Core i7-2760QM@2.40GHz (dated CPU for notebooks; time measured with simple code instrumentation, and using the throughput-sync.sh script of the S suite [1], in performance-profiling mode). To put this result into context, the total, single-lock-protected, per-request execution time of the lightest I/O scheduler available in blk-mq, mq-deadline, is 0.7 us (mq-deadline is ~800 LOC, against ~10500 LOC for BFQ).

Scheduling overhead further limits the maximum IOPS that a CPU can process (already limited by the execution of the rest of the I/O stack). To give an idea of the limits with BFQ, on slow or average CPUs, here are, first, the limits of BFQ for three different CPUs, on, respectively, an average laptop, an old desktop, and a cheap embedded system, in case full hierarchical support is enabled (i.e., CONFIG_BFQ_GROUP_IOSCHED is set), but CONFIG_BFQ_CGROUP_DEBUG is not set (Section 4-2): - Intel i7-4850HQ: 400 KIOPS - AMD A8-3850: 250 KIOPS - ARM CortexTM-A53 Octa-core: 80 KIOPS

If CONFIG_BFQ_CGROUP_DEBUG is set (and of course full hierarchical support is enabled), then the sustainable throughput with BFQ decreases, because all blkio.bfq* statistics are created and updated (Section 4-2). For BFQ, this leads to the following maximum sustainable throughputs, on the same systems as above:

- Intel i7-4850HQ: 310 KIOPS - AMD A8-3850: 200 KIOPS - ARM CortexTM-A53 Octa-core: 56 KIOPS

BFQ works for multi-queue devices too.

1.1 1. When may BFQ be useful?

BFQ provides the following benefits on personal and server systems.

1.1.1 1-1 Personal systems

Low latency for interactive applications

Regardless of the actual background workload, BFQ guarantees that, for interactive tasks, the storage device is virtually as responsive as if it was idle. For example, even if one or more of the following background workloads are being executed:

- one or more large files are being read, written or copied,
- a tree of source files is being compiled,
- one or more virtual machines are performing I/O,
- a software update is in progress,
- indexing daemons are scanning filesystems and updating their databases,

starting an application or loading a file from within an application takes about the same time as if the storage device was idle. As a comparison, with CFQ, NOOP or DEADLINE, and in the same conditions, applications experience high latencies, or even become unresponsive until the background workload terminates (also on SSDs).

Low latency for soft real-time applications

Also soft real-time applications, such as audio and video players/streamers, enjoy a low latency and a low drop rate, regardless of the background I/O workload. As a consequence, these applications do not suffer from almost any glitch due to the background workload.

Higher speed for code-development tasks

If some additional workload happens to be executed in parallel, then BFQ executes the I/O-related components of typical code-development tasks (compilation, checkout, merge, ⋯) much more quickly than CFQ, NOOP or DEADLINE.

High throughput

On hard disks, BFQ achieves up to 30% higher throughput than CFQ, and up to 150% higher throughput than DEADLINE and NOOP, with all the sequential workloads considered in our tests. With random workloads, and with all the workloads on flash-based devices, BFQ achieves, instead, about the same throughput as the other schedulers.

Strong fairness, bandwidth and delay guarantees

BFQ distributes the device throughput, and not just the device time, among I/O-bound applications in proportion their weights, with any workload and regardless of the device parameters. From these bandwidth guarantees, it is possible to compute tight per-I/O-request delay guarantees by a simple formula. If not configured for strict service guarantees, BFQ switches to time-based resource sharing (only) for applications that would otherwise cause a throughput loss.

1.1.2 1-2 Server systems

Most benefits for server systems follow from the same service properties as above. In particular, regardless of whether additional, possibly heavy workloads are being served, BFQ guarantees:

- audio and video-streaming with zero or very low jitter and drop rate;
- fast retrieval of WEB pages and embedded objects;
- real-time recording of data in live-dumping applications (e.g., packet logging);
- responsiveness in local and remote access to a server.

1.2 2. How does BFQ work?

BFQ is a proportional-share I/O scheduler, whose general structure, plus a lot of code, are borrowed from CFQ.

- Each process doing I/O on a device is associated with a weight and a *(bfq_)queue*.
- BFQ grants exclusive access to the device, for a while, to one queue (process) at a time, and implements this service model by associating every queue with a budget, measured in number of sectors.
  - After a queue is granted access to the device, the budget of the queue is decremented, on each request dispatch, by the size of the request.
  - The in-service queue is expired, i.e., its service is suspended, only if one of the following events occurs: 1) the queue finishes its budget, 2) the queue empties, 3) a "budget timeout" fires.
    - The budget timeout prevents processes doing random I/O from holding the device for too long and dramatically reducing throughput.

    * Actually, as in CFQ, a queue associated with a process issuing sync requests may not be expired immediately when it empties. In contrast, BFQ may idle the device for a short time interval, giving the process the chance to go on being served if it issues a new request in time. Device idling typically boosts the throughput on rotational devices and on non-queueing flash-based devices, if processes do synchronous and sequential I/O. In addition, under BFQ, device idling is also instrumental in guaranteeing the desired throughput fraction to processes issuing sync requests (see the description of the slice_idle tunable in this document, or [1, 2], for more details).

        · With respect to idling for service guarantees, if several processes are competing for the device at the same time, but all processes and groups have the same weight, then BFQ guarantees the expected throughput distribution without ever idling the device. Throughput is thus as high as possible in this common scenario.

    * On flash-based storage with internal queueing of commands (typically NCQ), device idling happens to be always detrimental for throughput. So, with these devices, BFQ performs idling only when strictly needed for service guarantees, i.e., for guaranteeing low latency or fairness. In these cases, overall throughput may be sub-optimal. No solution currently exists to provide both strong service guarantees and optimal throughput on devices with internal queueing.

- If low-latency mode is enabled (default configuration), BFQ executes some special heuristics to detect interactive and soft real-time applications (e.g., video or audio players/streamers), and to reduce their latency. The most important action taken to achieve this goal is to give to the queues associated with these applications more than their fair share of the device throughput. For brevity, we call just “weight-raising”the whole sets of actions taken by BFQ to privilege these queues. In particular, BFQ provides a milder form of weight-raising for interactive applications, and a stronger form for soft real-time applications.

- BFQ automatically deactivates idling for queues born in a burst of queue creations. In fact, these queues are usually associated with the processes of applications and services that benefit mostly from a high throughput. Examples are systemd during boot, or git grep.

- As CFQ, BFQ merges queues performing interleaved I/O, i.e., performing random I/O that becomes mostly sequential if merged. Differently from CFQ, BFQ achieves this goal with a more reactive mechanism, called Early Queue Merge (EQM). EQM is so responsive in detecting interleaved I/O (cooperating processes), that it enables BFQ to achieve a high throughput, by queue merging, even for queues for which CFQ needs a different mechanism, preemption, to get a high throughput. As such EQM is a unified mechanism to achieve a high throughput with interleaved I/O.

- Queues are scheduled according to a variant of WF2Q+, named B-WF2Q+, and implemented using an augmented rb-tree to preserve an O(log N) overall complexity. See [2] for more details. B-WF2Q+ is also ready for hierarchical scheduling, details in Section 4.

  - B-WF2Q+ guarantees a tight deviation with respect to an ideal, perfectly fair, and smooth service. In particular, B-WF2Q+ guarantees that each queue receives a fraction of the device throughput proportional to its weight, even if the throughput fluctuates, and regardless of: the device parameters, the current workload and the budgets assigned to the queue.
  - The last, budget-independence, property (although probably counterintuitive in the first place) is definitely beneficial, for the following reasons:
    * First, with any proportional-share scheduler, the maximum deviation with respect to an ideal service is proportional to the maximum budget (slice) assigned to queues. As a consequence, BFQ can keep this deviation tight not only because of the accurate service of B-WF2Q+, but also because BFQ *does not* need to assign a larger budget to a queue to let the queue receive a higher fraction of the device throughput.
    * Second, BFQ is free to choose, for every process (queue), the budget that best fits the needs of the process, or best leverages the I/O pattern of the process. In particular, BFQ updates queue budgets with a simple feedback-loop algorithm that allows a high throughput to be achieved, while still providing tight latency guarantees to time-sensitive applications. When the in-service queue expires, this algorithm computes the next budget of the queue so as to:
      · Let large budgets be eventually assigned to the queues associated with I/O-bound applications performing sequential I/O: in fact, the longer these applications are served once got access to the device, the higher the throughput is.
      · Let small budgets be eventually assigned to the queues associated with time-sensitive applications (which typically perform sporadic and short I/O), because, the smaller the budget assigned to a queue waiting for service is, the sooner B-WF2Q+ will serve that queue (Subsec 3.3 in [2]).

- If several processes are competing for the device at the same time, but all processes and groups have the same weight, then BFQ guarantees the expected throughput distribution without ever idling the device. It uses preemption instead. Throughput is then much higher in this common scenario.
- ioprio classes are served in strict priority order, i.e., lower-priority queues are not served as long as there are higher-priority queues. Among queues in the same class, the bandwidth is distributed in proportion to the weight of each queue. A very thin extra bandwidth is however guaranteed to the Idle class, to prevent it from starving.

1.3 3. What are BFQ's tunables and how to properly configure BFQ?

Most BFQ tunables affect service guarantees (basically latency and fairness) and throughput. For full details on how to choose the desired tradeoff between service guarantees and throughput, see the parameters slice_idle, strict_guarantees and low_latency. For details on how to maximise throughput, see slice_idle, timeout_sync and max_budget. The other performance-related parameters have been inherited from, and have been preserved mostly for compatibility with CFQ. So far, no performance improvement has been reported after changing the latter parameters in BFQ.

In particular, the tunables back_seek-max, back_seek_penalty, fifo_expire_async and fifo_expire_sync below are the same as in CFQ. Their description is just copied from that for CFQ. Some considerations in the description of slice_idle are copied from CFQ too.

1.3.1 per-process ioprio and weight

Unless the cgroups interface is used (see "4. BFQ group scheduling"), weights can be assigned to processes only indirectly, through I/O priorities, and according to the relation: weight = (IOPRIO_BE_NR - ioprio) * 10.

Beware that, if low-latency is set, then BFQ automatically raises the weight of the queues associated with interactive and soft real-time applications. Unset this tunable if you need/want to control weights.

1.3.2 slice_idle

This parameter specifies how long BFQ should idle for next I/O request, when certain sync BFQ queues become empty. By default slice_idle is a non-zero value. Idling has a double purpose: boosting throughput and making sure that the desired throughput distribution is respected (see the description of how BFQ works, and, if needed, the papers referred there).

As for throughput, idling can be very helpful on highly seeky media like single spindle SATA/SAS disks where we can cut down on overall number of seeks and see improved throughput.

Setting slice_idle to 0 will remove all the idling on queues and one should see an overall improved throughput on faster storage devices like multiple SATA/SAS disks in hardware RAID configuration, as well as flash-based storage with internal command queueing (and parallelism).

So depending on storage and workload, it might be useful to set slice_idle=0. In general for SATA/SAS disks and software RAID of SATA/SAS disks keeping slice_idle enabled should be useful. For any configurations where there are multiple spindles behind single LUN (Host based hardware RAID controller or for storage arrays), or with flash-based fast storage, setting slice_idle=0 might end up in better throughput and acceptable latencies.

Idling is however necessary to have service guarantees enforced in case of differentiated weights or differentiated I/O-request lengths. To see why, suppose that a given BFQ queue A must get several I/O requests served for each request served for another queue B. Idling ensures that, if A makes a new I/O request slightly after becoming empty, then no request of B is dispatched in the middle, and thus A does not lose the possibility to get more than one request dispatched before the next request of B is dispatched. Note that idling guarantees the desired differentiated treatment of queues only in terms of I/O-request dispatches. To guarantee that the actual service order then corresponds to the dispatch order, the strict_guarantees tunable must be set too.

There is an important flipside for idling: apart from the above cases where it is beneficial also for throughput, idling can severely impact throughput. One important case is random workload. Because of this issue, BFQ tends to avoid idling as much as possible, when it is not beneficial also for throughput (as detailed in Section 2). As a consequence of this behavior, and of further issues described for the strict_guarantees tunable, short-term service guarantees may be occasionally violated. And, in some cases, these guarantees may be more important than guaranteeing maximum throughput. For example, in video playing/streaming, a very low drop rate may be more important than maximum throughput. In these cases, consider setting the strict_guarantees parameter.

1.3.3 slice_idle_us

Controls the same tuning parameter as slice_idle, but in microseconds. Either tunable can be used to set idling behavior. Afterwards, the other tunable will reflect the newly set value in sysfs.

1.3.4 strict_guarantees

If this parameter is set (default: unset), then BFQ

- always performs idling when the in-service queue becomes empty;
- forces the device to serve one I/O request at a time, by dispatching a new request only if there is no outstanding request.

In the presence of differentiated weights or I/O-request sizes, both the above conditions are needed to guarantee that every BFQ queue receives its allotted share of the bandwidth. The first condition is needed for the reasons explained in the description of the slice_idle tunable. The second condition is needed because all modern storage devices reorder internally-queued requests, which may trivially break the service guarantees enforced by the I/O scheduler.

Setting strict_guarantees may evidently affect throughput.

1.3.5 back_seek_max

This specifies, given in Kbytes, the maximum "distance" for backward seeking. The distance is the amount of space from the current head location to the sectors that are backward in terms of distance.

This parameter allows the scheduler to anticipate requests in the "backward" direction and consider them as being the "next" if they are within this distance from the current head location.

1.3.6 back_seek_penalty

This parameter is used to compute the cost of backward seeking. If the backward distance of request is just 1/back_seek_penalty from a "front" request, then the seeking cost of two requests is considered equivalent.

So scheduler will not bias toward one or the other request (otherwise scheduler will bias toward front request). Default value of back_seek_penalty is 2.

1.3.7 fifo_expire_async

This parameter is used to set the timeout of asynchronous requests. Default value of this is 248ms.

1.3.8 fifo_expire_sync

This parameter is used to set the timeout of synchronous requests. Default value of this is 124ms. In case to favor synchronous requests over asynchronous one, this value should be decreased relative to fifo_expire_async.

1.3.9 low_latency

This parameter is used to enable/disable BFQ's low latency mode. By default, low latency mode is enabled. If enabled, interactive and soft real-time applications are privileged and experience a lower latency, as explained in more detail in the description of how BFQ works.

DISABLE this mode if you need full control on bandwidth distribution. In fact, if it is enabled, then BFQ automatically increases the bandwidth share of privileged applications, as the main means to guarantee a lower latency to them.

In addition, as already highlighted at the beginning of this document, DISABLE this mode if your only goal is to achieve a high throughput. In fact, privileging the I/O of some application over the rest may entail a lower throughput. To achieve the highest-possible throughput on a non-rotational device, setting slice_idle to 0 may be needed too (at the cost of giving up any strong guarantee on fairness and low latency).

1.3.10 timeout_sync

Maximum amount of device time that can be given to a task (queue) once it has been selected for service. On devices with costly seeks, increasing this time usually increases maximum throughput. On the opposite end, increasing this time coarsens the granularity of the short-term bandwidth and latency guarantees, especially if the following parameter is set to zero.

1.3.11 max_budget

Maximum amount of service, measured in sectors, that can be provided to a BFQ queue once it is set in service (of course within the limits of the above timeout). According to what said in the description of the algorithm, larger values increase the throughput in proportion to the percentage of sequential I/O requests issued. The price of larger values is that they coarsen the granularity of short-term bandwidth and latency guarantees.

The default value is 0, which enables auto-tuning: BFQ sets max_budget to the maximum number of sectors that can be served during timeout_sync, according to the estimated peak rate.

For specific devices, some users have occasionally reported to have reached a higher throughput by setting max_budget explicitly, i.e., by setting max_budget to a higher value than 0. In particular, they have set max_budget to higher values than those to which BFQ would have set it with auto-tuning. An alternative way to achieve this goal is to just increase the value of timeout_sync, leaving max_budget equal to 0.

1.4 4. Group scheduling with BFQ

BFQ supports both cgroups-v1 and cgroups-v2 io controllers, namely blkio and io. In particular, BFQ supports weight-based proportional share. To activate cgroups support, set BFQ_GROUP_IOSCHED.

1.4.1 4-1 Service guarantees provided

With BFQ, proportional share means true proportional share of the device bandwidth, according to group weights. For example, a group with weight 200 gets twice the bandwidth, and not just twice the time, of a group with weight 100.

BFQ supports hierarchies (group trees) of any depth. Bandwidth is distributed among groups and processes in the expected way: for each group, the children of the group share the whole bandwidth of the group in proportion to their weights. In particular, this implies that, for each leaf group, every process of the group receives the same share of the whole group bandwidth, unless the ioprio of the process is modified.

The resource-sharing guarantee for a group may partially or totally switch from bandwidth to time, if providing bandwidth guarantees to the group lowers the throughput too much. This switch occurs on a per-process basis: if a process of

a leaf group causes throughput loss if served in such a way to receive its share of the bandwidth, then BFQ switches back to just time-based proportional share for that process.

1.4.2 4-2 Interface

To get proportional sharing of bandwidth with BFQ for a given device, BFQ must of course be the active scheduler for that device.

Within each group directory, the names of the files associated with BFQ-specific cgroup parameters and stats begin with the "bfq." prefix. So, with cgroups-v1 or cgroups-v2, the full prefix for BFQ-specific files is "blkio.bfq." or "io.bfq." For example, the group parameter to set the weight of a group with BFQ is blkio.bfq.weight or io.bfq.weight.

As for cgroups-v1 (blkio controller), the exact set of stat files created, and kept up-to-date by bfq, depends on whether CONFIG_BFQ_CGROUP_DEBUG is set. If it is set, then bfq creates all the stat files documented in Documentation/admin-guide/cgroup-v1/blkio-controller.rst. If, instead, CONFIG_BFQ_CGROUP_DEBUG is not set, then bfq creates only the files:

```
blkio.bfq.io_service_bytes
blkio.bfq.io_service_bytes_recursive
blkio.bfq.io_serviced
blkio.bfq.io_serviced_recursive
```

The value of CONFIG_BFQ_CGROUP_DEBUG greatly influences the maximum throughput sustainable with bfq, because updating the blkio.bfq.* stats is rather costly, especially for some of the stats enabled by CONFIG_BFQ_CGROUP_DEBUG.

1.4.3 Parameters to set

For each group, there is only the following parameter to set.

weight (namely blkio.bfq.weight or io.bfq-weight): the weight of the group inside its parent. Available values: 1..1000 (default 100). The linear mapping between ioprio and weights, described at the beginning of the tunable section, is still valid, but all weights higher than IOPRIO_BE_NR*10 are mapped to ioprio 0.

Recall that, if low-latency is set, then BFQ automatically raises the weight of the queues associated with interactive and soft real-time applications. Unset this tunable if you need/want to control weights.

[1]

P. Valente, A. Avanzini, "Evolution of the BFQ Storage I/O Scheduler", Proceedings of the First Workshop on Mobile System Technologies (MST-2015), May 2015.

http://algogroup.unimore.it/people/paolo/disk_sched/mst-2015.pdf

[2]

P. Valente and M. Andreolini, "Improving Application Responsiveness with

the BFQ Disk I/O Scheduler”, Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR ‘12), June 2012.

Slightly extended version:

http://algogroup.unimore.it/people/paolo/disk_sched/bfq-v1-suite-results.pdf

[3]

https://github.com/Algodev-github/S

CHAPTER

TWO

NOTES ON THE GENERIC BLOCK LAYER REWRITE IN LINUX 2.5

Note: It seems that there are lot of outdated stuff here. This seems to be written somewhat as a task list. Yet, eventually, something here might still be useful.

Notes Written on Jan 15, 2002:

- Jens Axboe <jens.axboe@oracle.com>
- Suparna Bhattacharya <suparna@in.ibm.com>

Last Updated May 2, 2002

September 2003: Updated I/O Scheduler portions

- Nick Piggin <npiggin@kernel.dk>

2.1 Introduction

These are some notes describing some aspects of the 2.5 block layer in the context of the bio rewrite. The idea is to bring out some of the key changes and a glimpse of the rationale behind those changes.

Please mail corrections & suggestions to suparna@in.ibm.com.

2.2 Credits

2.5 bio rewrite:

- Jens Axboe <jens.axboe@oracle.com>

Many aspects of the generic block layer redesign were driven by and evolved over discussions, prior patches and the collective experience of several people. See sections 8 and 9 for a list of some related references.

The following people helped with review comments and inputs for this document:

- Christoph Hellwig <hch@infradead.org>
- Arjan van de Ven <arjanv@redhat.com>
- Randy Dunlap <rdunlap@xenotime.net>

- Andre Hedrick <andre@linux-ide.org>

The following people helped with fixes/contributions to the bio patches while it was still work-in-progress:

- David S. Miller <davem@redhat.com>

2.3 Bio Notes

Let us discuss the changes in the context of how some overall goals for the block layer are addressed.

2.4 1. Scope for tuning the generic logic to satisfy various requirements

The block layer design supports adaptable abstractions to handle common processing with the ability to tune the logic to an appropriate extent depending on the nature of the device and the requirements of the caller. One of the objectives of the rewrite was to increase the degree of tunability and to enable higher level code to utilize underlying device/driver capabilities to the maximum extent for better i/o performance. This is important especially in the light of ever improving hardware capabilities and application/middleware software designed to take advantage of these capabilities.

2.4.1 1.1 Tuning based on low level device / driver capabilities

Sophisticated devices with large built-in caches, intelligent i/o scheduling optimizations, high memory DMA support, etc may find some of the generic processing an overhead, while for less capable devices the generic functionality is essential for performance or correctness reasons. Knowledge of some of the capabilities or parameters of the device should be used at the generic block layer to take the right decisions on behalf of the driver.

How is this achieved ?

Tuning at a per-queue level:

i. Per-queue limits/values exported to the generic layer by the driver

Various parameters that the generic i/o scheduler logic uses are set at a per-queue level (e.g maximum request size, maximum number of segments in a scatter-gather list, logical block size)

Some parameters that were earlier available as global arrays indexed by major/minor are now directly associated with the queue. Some of these may move into the block device structure in the future. Some characteristics have been incorporated into a queue flags field rather than separate fields in themselves. There are blk_queue_xxx functions to set the parameters, rather than update the fields directly

Some new queue property settings:

blk_queue_bounce_limit(q, u64 dma_address)
Enable I/O to highmem pages, dma_address being the limit. No highmem default.

blk_queue_max_sectors(q, max_sectors)
Sets two variables that limit the size of the request.

- The request queue's max_sectors, which is a soft size in units of 512 byte sectors, and could be dynamically varied by the core kernel.
- The request queue's max_hw_sectors, which is a hard limit and reflects the maximum size request a driver can handle in units of 512 byte sectors.

The default for both max_sectors and max_hw_sectors is 255. The upper limit of max_sectors is 1024.

blk_queue_max_phys_segments(q, max_segments)
Maximum physical segments you can handle in a request. 128 default (driver limit). (See 3.2.2)

blk_queue_max_hw_segments(q, max_segments)
Maximum dma segments the hardware can handle in a request. 128 default (host adapter limit, after dma remapping). (See 3.2.2)

blk_queue_max_segment_size(q, max_seg_size)
Maximum size of a clustered segment, 64kB default.

blk_queue_logical_block_size(q, logical_block_size)
Lowest possible sector size that the hardware can operate on, 512 bytes default.

New queue flags:

- QUEUE_FLAG_CLUSTER (see 3.2.2)
- QUEUE_FLAG_QUEUED (see 3.2.4)

ii. High-mem i/o capabilities are now considered the default

The generic bounce buffer logic, present in 2.4, where the block layer would by default copyin/out i/o requests on high-memory buffers to low-memory buffers assuming that the driver wouldn't be able to handle it directly, has been changed in 2.5. The bounce logic is now applied only for memory ranges for which the device cannot handle i/o. A driver can specify this by setting the queue bounce limit for the request queue for the device (blk_queue_bounce_limit()). This avoids the inefficiencies of the copyin/out where a device is capable of handling high memory i/o.

In order to enable high-memory i/o where the device is capable of supporting it, the pci dma mapping routines and associated data structures have now been modified to accomplish a direct page -> bus translation, without requiring a virtual address mapping (unlike the earlier scheme of virtual address -> bus translation). So this works uniformly for high-memory pages (which do not have a corresponding kernel virtual address space mapping) and low-memory pages.

Note: Please refer to /core-api/dma-api-howto for a discussion on PCI high mem DMA aspects and mapping of scatter gather lists, and support for 64 bit PCI.

Special handling is required only for cases where i/o needs to happen on pages at physical memory addresses beyond what the device can support. In these cases, a bounce bio representing a buffer from the supported memory range is used for performing the i/o with copyin/copyout as needed depending on the type of the operation. For example, in case of a read operation, the data read has to be copied to the original buffer on i/o completion, so a callback routine is set up to do this, while for write, the data is copied from the original buffer to the bounce buffer prior to issuing the operation. Since an original buffer may be in a high memory area that's not mapped in kernel virtual addr, a kmap operation may be required for performing the copy, and special care may be needed in the completion path as it may not be in irq context. Special care is also required (by way of GFP flags) when allocating bounce buffers, to avoid certain highmem deadlock possibilities.

It is also possible that a bounce buffer may be allocated from high-memory area that's not mapped in kernel virtual addr, but within the range that the device can use directly; so the bounce page may need to be kmapped during copy operations. [Note: This does not hold in the current implementation, though]

There are some situations when pages from high memory may need to be kmapped, even if bounce buffers are not necessary. For example a device may need to abort DMA operations and revert to PIO for the transfer, in which case a virtual mapping of the page is required. For SCSI it is also done in some scenarios where the low level driver cannot be trusted to handle a single sg entry correctly. The driver is expected to perform the kmaps as needed on such occasions as appropriate. A driver could also use the blk_queue_bounce() routine on its own to bounce highmem i/o to low memory for specific requests if so desired.

iii. The i/o scheduler algorithm itself can be replaced/set as appropriate

As in 2.4, it is possible to plugin a brand new i/o scheduler for a particular queue or pick from (copy) existing generic schedulers and replace/override certain portions of it. The 2.5 rewrite provides improved modularization of the i/o scheduler. There are more pluggable callbacks, e.g for init, add request, extract request, which makes it possible to abstract specific i/o scheduling algorithm aspects and details outside of the generic loop. It also makes it possible to completely hide the implementation details of the i/o scheduler from block drivers.

I/O scheduler wrappers are to be used instead of accessing the queue directly. See section 4. The I/O scheduler for details.

2.4.2 1.2 Tuning Based on High level code capabilities

i. Application capabilities for raw i/o

This comes from some of the high-performance database/middleware requirements where an application prefers to make its own i/o scheduling decisions based on an understanding of the access patterns and i/o characteristics

ii. High performance filesystems or other higher level kernel code's capabilities

Kernel components like filesystems could also take their own i/o scheduling decisions for optimizing performance. Journalling filesystems may need some control over i/o ordering.

What kind of support exists at the generic block layer for this ?

The flags and rw fields in the bio structure can be used for some tuning from above e.g indicating that an i/o is just a readahead request, or priority settings (currently unused). As far as user applications are concerned they would need an additional mechanism either via open flags or ioctls, or some other upper level mechanism to communicate such settings to block.

1.2.1 Request Priority/Latency

Todo/Under discussion:

```
Arjan's proposed request priority scheme allows higher levels some␣
↪broad
control (high/med/low) over the priority  of an i/o request vs␣
↪other pending
requests in the queue. For example it allows reads for bringing in␣
↪an
executable page on demand to be given a higher priority over␣
↪pending write
requests which haven't aged too much on the queue. Potentially this␣
↪priority
could even be exposed to applications in some manner, providing␣
↪higher level
tunability. Time based aging avoids starvation of lower priority
requests. Some bits in the bi_opf flags field in the bio structure␣
↪are
intended to be used for this priority information.
```

2.4.3 1.3 Direct Access to Low level Device/Driver Capabilities (Bypass mode)

(e.g Diagnostics, Systems Management)

There are situations where high-level code needs to have direct access to the low level device capabilities or requires the ability to issue commands to the device bypassing some of the intermediate i/o layers. These could, for example, be special control commands issued through ioctl interfaces, or could be raw read/write commands that stress the drive’ s capabilities for certain kinds of fitness tests. Having direct interfaces at multiple levels without having to pass through upper layers makes it possible to perform bottom up validation of the i/o path, layer by layer, starting from the media.

The normal i/o submission interfaces, e.g submit_bio, could be bypassed for specially crafted requests which such ioctl or diagnostics interfaces would typically use, and the elevator add_request routine can instead be used to directly insert such requests in the queue or preferably the blk_do_rq routine can be used to place the request on the queue and wait for completion. Alternatively, sometimes the caller might just invoke a lower level driver specific interface with the request as a parameter.

If the request is a means for passing on special information associated with the command, then such information is associated with the request->special field

(rather than misuse the request->buffer field which is meant for the request data buffer' s virtual mapping).

For passing request data, the caller must build up a bio descriptor representing the concerned memory buffer if the underlying driver interprets bio segments or uses the block layer end*request* functions for i/o completion. Alternatively one could directly use the request->buffer field to specify the virtual address of the buffer, if the driver expects buffer addresses passed in this way and ignores bio entries for the request type involved. In the latter case, the driver would modify and manage the request->buffer, request->sector and request->nr_sectors or request->current_nr_sectors fields itself rather than using the block layer end_request or end_that_request_first completion interfaces. (See 2.3 or *struct request documentation* for a brief explanation of the request structure fields)

```
[TBD: end_that_request_last should be usable even in this case;
Perhaps an end_that_direct_request_first routine could be␣
↪implemented to make
handling direct requests easier for such drivers; Also for drivers␣
↪that
expect bios, a helper function could be provided for setting up a␣
↪bio
corresponding to a data buffer]

<JENS: I dont understand the above, why is end_that_request_first()␣
↪not
usable? Or _last for that matter. I must be missing something>

<SUP: What I meant here was that if the request doesn't have a bio,␣
↪then
 end_that_request_first doesn't modify nr_sectors or current_nr_
↪sectors,
 and hence can't be used for advancing request state settings on the
 completion of partial transfers. The driver has to modify these␣
↪fields
 directly by hand.
 This is because end_that_request_first only iterates over the bio␣
↪list,
 and always returns 0 if there are none associated with the request.
 _last works OK in this case, and is not a problem, as I mentioned␣
↪earlier
>
```

1.3.1 Pre-built Commands

A request can be created with a pre-built custom command to be sent directly to the device. The cmd block in the request structure has room for filling in the command bytes. (i.e rq->cmd is now 16 bytes in size, and meant for command pre-building, and the type of the request is now indicated through rq->flags instead of via rq->cmd)

The request structure flags can be set up to indicate the type of request in such cases (REQ_PC: direct packet command passed to driver, REQ_BLOCK_PC: packet command issued via blk_do_rq, REQ_SPECIAL: special request).

It can help to pre-build device commands for requests in advance. Drivers can now specify a request prepare function (q->prep_rq_fn) that the block layer would invoke to pre-build device commands for a given request, or perform other preparatory processing for the request. This is routine is called by elv_next_request(), i.e. typically just before servicing a request. (The prepare function would not be called for requests that have RQF_DONTPREP enabled)

Aside:

Pre-building could possibly even be done early, i.e before placing the request on the queue, rather than construct the command on the fly in the driver while servicing the request queue when it may affect latencies in interrupt context or responsiveness in general. One way to add early pre-building would be to do it whenever we fail to merge on a request. Now REQ_NOMERGE is set in the request flags to skip this one in the future, which means that it will not change before we feed it to the device. So the pre-builder hook can be invoked there.

2.5 2. Flexible and generic but minimalist i/o structure/descriptor

2.5.1 2.1 Reason for a new structure and requirements addressed

Prior to 2.5, buffer heads were used as the unit of i/o at the generic block layer, and the low level request structure was associated with a chain of buffer heads for a contiguous i/o request. This led to certain inefficiencies when it came to large i/o requests and readv/writev style operations, as it forced such requests to be broken up into small chunks before being passed on to the generic block layer, only to be merged by the i/o scheduler when the underlying device was capable of handling the i/o in one shot. Also, using the buffer head as an i/o structure for i/os that didn't originate from the buffer cache unnecessarily added to the weight of the descriptors which were generated for each such chunk.

The following were some of the goals and expectations considered in the redesign of the block i/o data structure in 2.5.

1. Should be appropriate as a descriptor for both raw and buffered i/o - avoid cache related fields which are irrelevant in the direct/page i/o path, or filesystem block size alignment restrictions which may not be relevant for raw i/o.

2. Ability to represent high-memory buffers (which do not have a virtual address mapping in kernel address space).
3. Ability to represent large i/os w/o unnecessarily breaking them up (i.e greater than PAGE_SIZE chunks in one shot)
4. At the same time, ability to retain independent identity of i/os from different sources or i/o units requiring individual completion (e.g. for latency reasons)
5. Ability to represent an i/o involving multiple physical memory segments (including non-page aligned page fragments, as specified via readv/writev) without unnecessarily breaking it up, if the underlying device is capable of handling it.
6. Preferably should be based on a memory descriptor structure that can be passed around different types of subsystems or layers, maybe even networking, without duplication or extra copies of data/descriptor fields themselves in the process
7. Ability to handle the possibility of splits/merges as the structure passes through layered drivers (lvm, md, evms), with minimal overhead.

The solution was to define a new structure (bio) for the block layer, instead of using the buffer head structure (bh) directly, the idea being avoidance of some associated baggage and limitations. The bio structure is uniformly used for all i/o at the block layer ; it forms a part of the bh structure for buffered i/o, and in the case of raw/direct i/o kiobufs are mapped to bio structures.

2.5.2 2.2 The bio struct

The bio structure uses a vector representation pointing to an array of tuples of <page, offset, len> to describe the i/o buffer, and has various other fields describing i/o parameters and state that needs to be maintained for performing the i/o.

Notice that this representation means that a bio has no virtual address mapping at all (unlike buffer heads).

```
struct bio_vec {
     struct page     *bv_page;
     unsigned short  bv_len;
     unsigned short  bv_offset;
};

/*
 * main unit of I/O for the block layer and lower layers (ie␣
↪drivers)
 */
struct bio {
     struct bio          *bi_next;    /* request queue link */
     struct block_device *bi_bdev;    /* target device */
     unsigned long       bi_flags;    /* status, command, etc */
     unsigned long       bi_opf;       /* low bits: r/w, high:␣
↪priority */
```

(continues on next page)

(continued from previous page)

```
    unsigned int     bi_vcnt;     /* how may bio_vec's */
    struct bvec_iter bi_iter;        /* current index into bio_vec␣
↪array */

    unsigned int     bi_size;     /* total size in bytes */
    unsigned short   bi_hw_segments; /* segments after DMA␣
↪remapping */
    unsigned int     bi_max;      /* max bio_vecs we can hold
                                     used as index into pool */
    struct bio_vec   *bi_io_vec;  /* the actual vec list */
    bio_end_io_t     *bi_end_io;  /* bi_end_io (bio) */
    atomic_t         bi_cnt;      /* pin count: free when it hits␣
↪zero */
    void             *bi_private;
};
```

With this multipage bio design:

- Large i/os can be sent down in one go using a bio_vec list consisting of an array of <page, offset, len> fragments (similar to the way fragments are represented in the zero-copy network code)
- Splitting of an i/o request across multiple devices (as in the case of lvm or raid) is achieved by cloning the bio (where the clone points to the same bi_io_vec array, but with the index and size accordingly modified)
- A linked list of bios is used as before for unrelated merges[1] - this avoids reallocs and makes independent completions easier to handle.
- Code that traverses the req list can find all the segments of a bio by using rq_for_each_segment. This handles the fact that a request has multiple bios, each of which can have multiple segments.
- Drivers which can't process a large bio in one shot can use the bi_iter field to keep track of the next bio_vec entry to process. (e.g a 1MB bio_vec needs to be handled in max 128kB chunks for IDE) [TBD: Should preferably also have a bi_voffset and bi_vlen to avoid modifying bi_offset an len fields]

bi_end_io() i/o callback gets called on i/o completion of the entire bio.

At a lower level, drivers build a scatter gather list from the merged bios. The scatter gather list is in the form of an array of <page, offset, len> entries with their corresponding dma address mappings filled in at the appropriate time. As an optimization, contiguous physical pages can be covered by a single entry where <page> refers to the first page and <len> covers the range of pages (up to 16 contiguous pages could be covered this way). There is a helper routine (blk_rq_map_sg) which drivers can use to build the sg list.

Note: Right now the only user of bios with more than one page is ll_rw_kio, which in turn means that only raw I/O uses it (direct i/o may not work right now). The intent however is to enable clustering of pages etc to become possible. The pagebuf

[1] unrelated merges – a request ends up containing two or more bios that didn't originate from the same place.

abstraction layer from SGI also uses multi-page bios, but that is currently not included in the stock development kernels. The same is true of Andrew Morton's work-in-progress multipage bio writeout and readahead patches.

2.5.3 2.3 Changes in the Request Structure

The request structure is the structure that gets passed down to low level drivers. The block layer make_request function builds up a request structure, places it on the queue and invokes the drivers request_fn. The driver makes use of block layer helper routine elv_next_request to pull the next request off the queue. Control or diagnostic functions might bypass block and directly invoke underlying driver entry points passing in a specially constructed request structure.

Only some relevant fields (mainly those which changed or may be referred to in some of the discussion here) are listed below, not necessarily in the order in which they occur in the structure (see include/linux/blkdev.h) Refer to *struct request documentation* for details about all the request structure fields and a quick reference about the layers which are supposed to use or modify those fields:

```
struct request {
      struct list_head queuelist;  /* Not meant to be directly␣
↪accessed by
                                      the driver.
                                      Used by q->elv_next_request_fn
                                      rq->queue is gone
                                      */
      .
      .
      unsigned char cmd[16]; /* prebuilt command data block */
      unsigned long flags;   /* also includes earlier rq->cmd␣
↪settings */
      .
      .
      sector_t sector; /* this field is now of type sector_t␣
↪instead of int
                          preparation for 64 bit sectors */
      .
      .

      /* Number of scatter-gather DMA addr+len pairs after
       * physical address coalescing is performed.
       */
      unsigned short nr_phys_segments;

      /* Number of scatter-gather addr+len pairs after
       * physical and DMA remapping hardware coalescing is␣
↪performed.
       * This is the number of scatter-gather entries the driver
       * will actually have to deal with after DMA mapping is done.
       */
      unsigned short nr_hw_segments;
```

(continues on next page)

(continued from previous page)

```
     /* Various sector counts */
     unsigned long nr_sectors;  /* no. of sectors left: driver␣
↪modifiable */
     unsigned long hard_nr_sectors;  /* block internal copy of␣
↪above */
     unsigned int current_nr_sectors; /* no. of sectors left in the
                                         current segment:driver␣
↪modifiable */
     unsigned long hard_cur_sectors; /* block internal copy of the␣
↪above */
     .
     .
     int tag;        /* command tag associated with request */
     void *special;  /* same as before */
     char *buffer;   /* valid only for low memory buffers up to
                      current_nr_sectors */
     .
     .
     struct bio *bio, *biotail;  /* bio list instead of bh */
     struct request_list *rl;
}
```

See the req_ops and req_flag_bits definitions for an explanation of the various flags available. Some bits are used by the block layer or i/o scheduler.

The behaviour of the various sector counts are almost the same as before, except that since we have multi-segment bios, current_nr_sectors refers to the numbers of sectors in the current segment being processed which could be one of the many segments in the current bio (i.e i/o completion unit). The nr_sectors value refers to the total number of sectors in the whole request that remain to be transferred (no change). The purpose of the hard_xxx values is for block to remember these counts every time it hands over the request to the driver. These values are updated by block on end_that_request_first, i.e. every time the driver completes a part of the transfer and invokes block end*request helpers to mark this. The driver should not modify these values. The block layer sets up the nr_sectors and current_nr_sectors fields (based on the corresponding hard_xxx values and the number of bytes transferred) and updates it on every transfer that invokes end_that_request_first. It does the same for the buffer, bio, bio->bi_iter fields too.

The buffer field is just a virtual address mapping of the current segment of the i/o buffer in cases where the buffer resides in low-memory. For high memory i/o, this field is not valid and must not be used by drivers.

Code that sets up its own request structures and passes them down to a driver needs to be careful about interoperation with the block layer helper functions which the driver uses. (Section 1.3)

2.6 3. Using bios

2.6.1 3.1 Setup/Teardown

There are routines for managing the allocation, and reference counting, and freeing of bios (bio_alloc, bio_get, bio_put).

This makes use of Ingo Molnar's mempool implementation, which enables subsystems like bio to maintain their own reserve memory pools for guaranteed deadlock-free allocations during extreme VM load. For example, the VM subsystem makes use of the block layer to writeout dirty pages in order to be able to free up memory space, a case which needs careful handling. The allocation logic draws from the preallocated emergency reserve in situations where it cannot allocate through normal means. If the pool is empty and it can wait, then it would trigger action that would help free up memory or replenish the pool (without deadlocking) and wait for availability in the pool. If it is in IRQ context, and hence not in a position to do this, allocation could fail if the pool is empty. In general mempool always first tries to perform allocation without having to wait, even if it means digging into the pool as long it is not less that 50% full.

On a free, memory is released to the pool or directly freed depending on the current availability in the pool. The mempool interface lets the subsystem specify the routines to be used for normal alloc and free. In the case of bio, these routines make use of the standard slab allocator.

The caller of bio_alloc is expected to taken certain steps to avoid deadlocks, e.g. avoid trying to allocate more memory from the pool while already holding memory obtained from the pool.

```
[TBD: This is a potential issue, though a rare possibility
 in the bounce bio allocation that happens in the current code, since
 it ends up allocating a second bio from the same pool while
 holding the original bio ]
```

Memory allocated from the pool should be released back within a limited amount of time (in the case of bio, that would be after the i/o is completed). This ensures that if part of the pool has been used up, some work (in this case i/o) must already be in progress and memory would be available when it is over. If allocating from multiple pools in the same code path, the order or hierarchy of allocation needs to be consistent, just the way one deals with multiple locks.

The bio_alloc routine also needs to allocate the bio_vec_list (bvec_alloc()) for a non-clone bio. There are the 6 pools setup for different size biovecs, so bio_alloc(gfp_mask, nr_iovecs) will allocate a vec_list of the given size from these slabs.

The bio_get() routine may be used to hold an extra reference on a bio prior to i/o submission, if the bio fields are likely to be accessed after the i/o is issued (since the bio may otherwise get freed in case i/o completion happens in the meantime).

The bio_clone_fast() routine may be used to duplicate a bio, where the clone shares the bio_vec_list with the original bio (i.e. both point to the same bio_vec_list). This would typically be used for splitting i/o requests in lvm or md.

2.6.2 3.2 Generic bio helper Routines

3.2.1 Traversing segments and completion units in a request

The macro rq_for_each_segment() should be used for traversing the bios in the request list (drivers should avoid directly trying to do it themselves). Using these helpers should also make it easier to cope with block changes in the future.

```
struct req_iterator iter;
rq_for_each_segment(bio_vec, rq, iter)
        /* bio_vec is now current segment */
```

I/O completion callbacks are per-bio rather than per-segment, so drivers that traverse bio chains on completion need to keep that in mind. Drivers which don't make a distinction between segments and completion units would need to be reorganized to support multi-segment bios.

3.2.2 Setting up DMA scatterlists

The blk_rq_map_sg() helper routine would be used for setting up scatter gather lists from a request, so a driver need not do it on its own.

    nr_segments = blk_rq_map_sg(q, rq, scatterlist);

The helper routine provides a level of abstraction which makes it easier to modify the internals of request to scatterlist conversion down the line without breaking drivers. The blk_rq_map_sg routine takes care of several things like collapsing physically contiguous segments (if QUEUE_FLAG_CLUSTER is set) and correct segment accounting to avoid exceeding the limits which the i/o hardware can handle, based on various queue properties.

- Prevents a clustered segment from crossing a 4GB mem boundary
- Avoids building segments that would exceed the number of physical memory segments that the driver can handle (phys_segments) and the number that the underlying hardware can handle at once, accounting for DMA remapping (hw_segments) (i.e. IOMMU aware limits).

Routines which the low level driver can use to set up the segment limits:

blk_queue_max_hw_segments() : Sets an upper limit of the maximum number of hw data segments in a request (i.e. the maximum number of address/length pairs the host adapter can actually hand to the device at once)

blk_queue_max_phys_segments() : Sets an upper limit on the maximum number of physical data segments in a request (i.e. the largest sized scatter list a driver could handle)

3.2.3 I/O completion

The existing generic block layer helper routines end_request, end_that_request_first and end_that_request_last can be used for i/o completion (and setting things up so the rest of the i/o or the next request can be kicked of) as before. With the introduction of multi-page bio support, end_that_request_first requires an additional argument indicating the number of sectors completed.

3.2.4 Implications for drivers that do not interpret bios

(don't handle multiple segments)

Drivers that do not interpret bios e.g those which do not handle multiple segments and do not support i/o into high memory addresses (require bounce buffers) and expect only virtually mapped buffers, can access the rq->buffer field. As before the driver should use current_nr_sectors to determine the size of remaining data in the current segment (that is the maximum it can transfer in one go unless it interprets segments), and rely on the block layer end_request, or end_that_request_first/last to take care of all accounting and transparent mapping of the next bio segment when a segment boundary is crossed on completion of a transfer. (The end*request* functions should be used if only if the request has come down from block/bio path, not for direct access requests which only specify rq->buffer without a valid rq->bio)

2.6.3 3.3 I/O Submission

The routine submit_bio() is used to submit a single io. Higher level i/o routines make use of this:

(a) Buffered i/o:

The routine submit_bh() invokes submit_bio() on a bio corresponding to the bh, allocating the bio if required. ll_rw_block() uses submit_bh() as before.

(b) Kiobuf i/o (for raw/direct i/o):

The ll_rw_kio() routine breaks up the kiobuf into page sized chunks and maps the array to one or more multi-page bios, issuing submit_bio() to perform the i/o on each of these.

The embedded bh array in the kiobuf structure has been removed and no preallocation of bios is done for kiobufs. [The intent is to remove the blocks array as well, but it's currently in there to kludge around direct i/o.] Thus kiobuf allocation has switched back to using kmalloc rather than vmalloc.

Todo/Observation:

> A single kiobuf structure is assumed to correspond to a contiguous range of data, so brw_kiovec() invokes ll_rw_kio for each kiobuf in a kiovec. So right now it wouldn't work for direct i/o on non-contiguous blocks. This is to be resolved. The eventual direction is to replace kiobuf by kvec's.
>
> Badari Pulavarty has a patch to implement direct i/o correctly using bio and kvec.

(c) Page i/o:

Todo/Under discussion:

> Andrew Morton's multi-page bio patches attempt to issue multi-page writeouts (and reads) from the page cache, by directly building up large bios for submission completely bypassing the usage of buffer heads. This work is still in progress.
>
> Christoph Hellwig had some code that uses bios for page-io (rather than bh). This isn't included in bio as yet. Christoph was also working on a design for representing virtual/real extents as an entity and modifying some of the address space ops interfaces to utilize this abstraction rather than buffer_heads. (This is somewhat along the lines of the SGI XFS pagebuf abstraction, but intended to be as lightweight as possible).

(d) Direct access i/o:

Direct access requests that do not contain bios would be submitted differently as discussed earlier in section 1.3.

Aside:

> Kvec i/o:
>
> Ben LaHaise's aio code uses a slightly different structure instead of kiobufs, called a kvec_cb. This contains an array of <page, offset, len> tuples (very much like the networking code), together with a callback function and data pointer. This is embedded into a brw_cb structure when passed to brw_kvec_async().
>
> Now it should be possible to directly map these kvecs to a bio. Just as while cloning, in this case rather than PRE_BUILT bio_vecs, we set the bi_io_vec array pointer to point to the veclet array in kvecs.
>
> TBD: In order for this to work, some changes are needed in the way multi-page bios are handled today. The values of the tuples in such a vector passed in from higher level code should not be modified by the block layer in the course of its request processing, since that would make it hard for the higher layer to continue to use the vector descriptor (kvec) after i/o completes. Instead, all such transient state should either be maintained in the request structure, and passed on in some way to the endio completion routine.

2.7 4. The I/O scheduler

I/O scheduler, a.k.a. elevator, is implemented in two layers. Generic dispatch queue and specific I/O schedulers. Unless stated otherwise, elevator is used to refer to both parts and I/O scheduler to specific I/O schedulers.

Block layer implements generic dispatch queue in *block/*.c*. The generic dispatch queue is responsible for requeueing, handling non-fs requests and all other subtleties.

Specific I/O schedulers are responsible for ordering normal filesystem requests. They can also choose to delay certain requests to improve throughput or whatever

purpose. As the plural form indicates, there are multiple I/O schedulers. They can be built as modules but at least one should be built inside the kernel. Each queue can choose different one and can also change to another one dynamically.

A block layer call to the i/o scheduler follows the convention elv_xxx(). This calls elevator_xxx_fn in the elevator switch (block/elevator.c). Oh, xxx and xxx might not match exactly, but use your imagination. If an elevator doesn't implement a function, the switch does nothing or some minimal house keeping work.

2.7.1 4.1. I/O scheduler API

The functions an elevator may implement are: (* are mandatory)

elevator_m	called to query requests for merge with a bio
elevator_m	called when two requests get merged. the one which gets merged into the other one will be never seen by I/O scheduler again. IOW, after being merged, the request is gone.
elevator_m	called when a request in the scheduler has been involved in a merge. It is used in the deadline scheduler for example, to reposition the request if its sorting order has changed.
elevator_al	called whenever the block layer determines that a bio can be merged into an existing request safely. The io scheduler may still want to stop a merge at this point if it results in some sort of conflict internally, this hook allows it to do that. Note however that two *requests* can still be merged at later time. Currently the io scheduler has no way to prevent that. It can only learn about the fact from elevator_merge_req_fn callback.
elevator_di	fills the dispatch queue with ready requests. I/O schedulers are free to postpone requests by not filling the dispatch queue unless @force is non-zero. Once dispatched, I/O schedulers are not allowed to manipulate the requests - they belong to generic dispatch queue.
elevator_ac	called to add a new request into the scheduler
elevator_fo	
elevator_la	These return the request before or after the one specified in disk sort order. Used by the block layer to find merge possibilities.
elevator_cc	called when a request is completed.
elevator_se	
elevator_pu	Must be used to allocate and free any elevator specific storage for a request.
elevator_ac	Called when device driver first sees a request. I/O schedulers can use this callback to determine when actual execution of a request starts.
elevator_de	Called when device driver decides to delay a request by requeueing it.
elevator_in	
elevator_ex	Allocate and free any elevator specific storage for a queue.

2.7.2 4.2 Request flows seen by I/O schedulers

All requests seen by I/O schedulers strictly follow one of the following three flows.

set_req_fn ->

i. add_req_fn -> (merged_fn ->)* -> dispatch_fn -> activate_req_fn -> (deactivate_req_fn -> activate_req_fn ->)* -> completed_req_fn

ii. add_req_fn -> (merged_fn ->)* -> merge_req_fn

iii. [none]

-> put_req_fn

2.7.3 4.3 I/O scheduler implementation

The generic i/o scheduler algorithm attempts to sort/merge/batch requests for optimal disk scan and request servicing performance (based on generic principles and device capabilities), optimized for:

i. improved throughput

ii. improved latency

iii. better utilization of h/w & CPU time

Characteristics:

i. Binary tree AS and deadline i/o schedulers use red black binary trees for disk position sorting and searching, and a fifo linked list for time-based searching. This gives good scalability and good availability of information. Requests are almost always dispatched in disk sort order, so a cache is kept of the next request in sort order to prevent binary tree lookups.

This arrangement is not a generic block layer characteristic however, so elevators may implement queues as they please.

ii. Merge hash AS and deadline use a hash table indexed by the last sector of a request. This enables merging code to quickly look up "back merge" candidates, even when multiple I/O streams are being performed at once on one disk.

"Front merges", a new request being merged at the front of an existing request, are far less common than "back merges" due to the nature of most I/O patterns. Front merges are handled by the binary trees in AS and deadline schedulers.

iii. Plugging the queue to batch requests in anticipation of opportunities for merge/sort optimizations

Plugging is an approach that the current i/o scheduling algorithm resorts to so that it collects up enough requests in the queue to be able to take advantage of the sorting/merging logic in the elevator. If the queue is empty when a request comes in, then it plugs the request queue (sort of like plugging the bath tub of a vessel to get fluid to build up) till it fills up with a few more requests, before starting to service the requests. This provides an opportunity to merge/sort the requests before passing them down to the device. There are various conditions when the queue is unplugged (to open up the flow again), either through a scheduled task or could be on demand. For example wait_on_buffer sets the unplugging going through

sync_buffer() running blk_run_address_space(mapping). Or the caller can do it explicity through blk_unplug(bdev). So in the read case, the queue gets explicitly unplugged as part of waiting for completion on that buffer.

Aside:

This is kind of controversial territory, as it's not clear if plugging is always the right thing to do. Devices typically have their own queues, and allowing a big queue to build up in software, while letting the device be idle for a while may not always make sense. The trick is to handle the fine balance between when to plug and when to open up. Also now that we have multi-page bios being queued in one shot, we may not need to wait to merge a big request from the broken up pieces coming by.

2.7.4 4.4 I/O contexts

I/O contexts provide a dynamically allocated per process data area. They may be used in I/O schedulers, and in the block layer (could be used for IO statis, priorities for example). See **io_context* in block/ll_rw_blk.c, and as-iosched.c for an example of usage in an i/o scheduler.

2.8 5. Scalability related changes

2.8.1 5.1 Granular Locking: io_request_lock replaced by a per-queue lock

The global io_request_lock has been removed as of 2.5, to avoid the scalability bottleneck it was causing, and has been replaced by more granular locking. The request queue structure has a pointer to the lock to be used for that queue. As a result, locking can now be per-queue, with a provision for sharing a lock across queues if necessary (e.g the scsi layer sets the queue lock pointers to the corresponding adapter lock, which results in a per host locking granularity). The locking semantics are the same, i.e. locking is still imposed by the block layer, grabbing the lock before request_fn execution which it means that lots of older drivers should still be SMP safe. Drivers are free to drop the queue lock themselves, if required. Drivers that explicitly used the io_request_lock for serialization need to be modified accordingly. Usually it's as easy as adding a global lock:

```
static DEFINE_SPINLOCK(my_driver_lock);
```

and passing the address to that lock to blk_init_queue().

2.8.2 5.2 64 bit sector numbers (sector_t prepares for 64 bit support)

The sector number used in the bio structure has been changed to sector_t, which could be defined as 64 bit in preparation for 64 bit sector support.

2.9 6. Other Changes/Implications

2.9.1 6.1 Partition re-mapping handled by the generic block layer

In 2.5 some of the gendisk/partition related code has been reorganized. Now the generic block layer performs partition-remapping early and thus provides drivers with a sector number relative to whole device, rather than having to take partition number into account in order to arrive at the true sector number. The routine blk_partition_remap() is invoked by submit_bio_noacct even before invoking the queue specific ->submit_bio, so the i/o scheduler also gets to operate on whole disk sector numbers. This should typically not require changes to block drivers, it just never gets to invoke its own partition sector offset calculations since all bios sent are offset from the beginning of the device.

2.10 7. A Few Tips on Migration of older drivers

Old-style drivers that just use CURRENT and ignores clustered requests, may not need much change. The generic layer will automatically handle clustered requests, multi-page bios, etc for the driver.

For a low performance driver or hardware that is PIO driven or just doesn't support scatter-gather changes should be minimal too.

The following are some points to keep in mind when converting old drivers to bio.

Drivers should use elv_next_request to pick up requests and are no longer supposed to handle looping directly over the request list. (struct request->queue has been removed)

Now end_that_request_first takes an additional number_of_sectors argument. It used to handle always just the first buffer_head in a request, now it will loop and handle as many sectors (on a bio-segment granularity) as specified.

Now bh->b_end_io is replaced by bio->bi_end_io, but most of the time the right thing to use is bio_endio(bio) instead.

If the driver is dropping the io_request_lock from its request_fn strategy, then it just needs to replace that with q->queue_lock instead.

As described in Sec 1.1, drivers can set max sector size, max segment size etc per queue now. Drivers that used to define their own merge functions i to handle things like this can now just use the blk_queue_* functions at blk_init_queue time.

Drivers no longer have to map a {partition, sector offset} into the correct absolute location anymore, this is done by the block layer, so where a driver received a request ala this before:

```
rq->rq_dev = mk_kdev(3, 5);     /* /dev/hda5 */
rq->sector = 0;                 /* first sector on hda5 */
```

it will now see:

```
rq->rq_dev = mk_kdev(3, 0);     /* /dev/hda */
rq->sector = 123128;            /* offset from start of disk */
```

As mentioned, there is no virtual mapping of a bio. For DMA, this is not a problem as the driver probably never will need a virtual mapping. Instead it needs a bus mapping (dma_map_page for a single segment or use dma_map_sg for scatter gather) to be able to ship it to the driver. For PIO drivers (or drivers that need to revert to PIO transfer once in a while (IDE for example)), where the CPU is doing the actual data transfer a virtual mapping is needed. If the driver supports highmem I/O, (Sec 1.1, (ii)) it needs to use kmap_atomic or similar to temporarily map a bio into the virtual address space.

2.11 8. Prior/Related/Impacted patches

2.11.1 8.1. Earlier kiobuf patches (sct/axboe/chait/hch/mkp)

- orig kiobuf & raw i/o patches (now in 2.4 tree)
- direct kiobuf based i/o to devices (no intermediate bh's)
- page i/o using kiobuf
- kiobuf splitting for lvm (mkp)
- elevator support for kiobuf request merging (axboe)

2.11.2 8.2. Zero-copy networking (Dave Miller)

2.11.3 8.3. SGI XFS - pagebuf patches - use of kiobufs

2.11.4 8.4. Multi-page pioent patch for bio (Christoph Hellwig)

2.11.5 8.5. Direct i/o implementation (Andrea Arcangeli) since 2.4.10-pre11

2.11.6 8.6. Async i/o implementation patch (Ben LaHaise)

2.11.7 8.7. EVMS layering design (IBM EVMS team)

2.11.8 8.8. Larger page cache size patch (Ben LaHaise) and Large page size (Daniel Phillips)

=> larger contiguous physical memory buffers

2.11.9 8.9. VM reservations patch (Ben LaHaise)

2.11.10 8.10. Write clustering patches ? (Marcelo/Quintela/Riel ?)

2.11.11 8.11. Block device in page cache patch (Andrea Archangeli) - now in 2.4.10+

2.11.12 8.12. Multiple block-size transfers for faster raw i/o (Shailabh Nagar, Badari)

2.11.13 8.13 Priority based i/o scheduler - prepatches (Arjan van de Ven)

2.11.14 8.14 IDE Taskfile i/o patch (Andre Hedrick)

2.11.15 8.15 Multi-page writeout and readahead patches (Andrew Morton)

2.11.16 8.16 Direct i/o patches for 2.5 using kvec and bio (Badari Pulavarthy)

2.12 9. Other References

2.12.1 9.1 The Splice I/O Model

Larry McVoy (and subsequent discussions on lkml, and Linus' comments - Jan 2001

2.12.2 9.2 Discussions about kiobuf and bh design

On lkml between sct, linus, alan et al - Feb-March 2001 (many of the initial thoughts that led to bio were brought up in this discussion thread)

2.12.3 9.3 Discussions on mempool on lkml - Dec 2001.

CHAPTER

THREE

IMMUTABLE BIOVECS AND BIOVEC ITERATORS

Kent Overstreet <kmo@daterainc.com>

As of 3.13, biovecs should never be modified after a bio has been submitted. Instead, we have a new struct bvec_iter which represents a range of a biovec - the iterator will be modified as the bio is completed, not the biovec.

More specifically, old code that needed to partially complete a bio would update bi_sector and bi_size, and advance bi_idx to the next biovec. If it ended up partway through a biovec, it would increment bv_offset and decrement bv_len by the number of bytes completed in that biovec.

In the new scheme of things, everything that must be mutated in order to partially complete a bio is segregated into struct bvec_iter: bi_sector, bi_size and bi_idx have been moved there; and instead of modifying bv_offset and bv_len, struct bvec_iter has bi_bvec_done, which represents the number of bytes completed in the current bvec.

There are a bunch of new helper macros for hiding the gory details - in particular, presenting the illusion of partially completed biovecs so that normal code doesn't have to deal with bi_bvec_done.

- Driver code should no longer refer to biovecs directly; we now have bio_iovec() and bio_iter_iovec() macros that return literal struct biovecs, constructed from the raw biovecs but taking into account bi_bvec_done and bi_size.

  bio_for_each_segment() has been updated to take a bvec_iter argument instead of an integer (that corresponded to bi_idx); for a lot of code the conversion just required changing the types of the arguments to bio_for_each_segment().

- Advancing a bvec_iter is done with bio_advance_iter(); bio_advance() is a wrapper around bio_advance_iter() that operates on bio->bi_iter, and also advances the bio integrity's iter if present.

  There is a lower level advance function - bvec_iter_advance() - which takes a pointer to a biovec, not a bio; this is used by the bio integrity code.

3.1 What's all this get us?

Having a real iterator, and making biovecs immutable, has a number of advantages:

- Before, iterating over bios was very awkward when you weren't processing exactly one bvec at a time - for example, bio_copy_data() in block/bio.c, which copies the contents of one bio into another. Because the biovecs wouldn't necessarily be the same size, the old code was tricky convoluted - it had to walk two different bios at the same time, keeping both bi_idx and and offset into the current biovec for each.

  The new code is much more straightforward - have a look. This sort of pattern comes up in a lot of places; a lot of drivers were essentially open coding bvec iterators before, and having common implementation considerably simplifies a lot of code.

- Before, any code that might need to use the biovec after the bio had been completed (perhaps to copy the data somewhere else, or perhaps to resubmit it somewhere else if there was an error) had to save the entire bvec array - again, this was being done in a fair number of places.

- Biovecs can be shared between multiple bios - a bvec iter can represent an arbitrary range of an existing biovec, both starting and ending midway through biovecs. This is what enables efficient splitting of arbitrary bios. Note that this means we _only_ use bi_size to determine when we've reached the end of a bio, not bi_vcnt - and the bio_iovec() macro takes bi_size into account when constructing biovecs.

- Splitting bios is now much simpler. The old bio_split() didn't even work on bios with more than a single bvec! Now, we can efficiently split arbitrary size bios - because the new bio can share the old bio's biovec.

  Care must be taken to ensure the biovec isn't freed while the split bio is still using it, in case the original bio completes first, though. Using bio_chain() when splitting bios helps with this.

- Submitting partially completed bios is now perfectly fine - this comes up occasionally in stacking block drivers and various code (e.g. md and bcache) had some ugly workarounds for this.

  It used to be the case that submitting a partially completed bio would work fine to _most_ devices, but since accessing the raw bvec array was the norm, not all drivers would respect bi_idx and those would break. Now, since all drivers _must_ go through the bvec iterator - and have been audited to make sure they are - submitting partially completed bios is perfectly fine.

3.2 Other implications:

- Almost all usage of bi_idx is now incorrect and has been removed; instead, where previously you would have used bi_idx you'd now use a bvec_iter, probably passing it to one of the helper macros.

  I.e. instead of using bio_iovec_idx() (or bio->bi_iovec[bio->bi_idx]), you now use bio_iter_iovec(), which takes a bvec_iter and returns a literal struct bio_vec - constructed on the fly from the raw biovec but taking into account bi_bvec_done (and bi_size).

- bi_vcnt can't be trusted or relied upon by driver code - i.e. anything that doesn't actually own the bio. The reason is twofold: firstly, it's not actually needed for iterating over the bio anymore - we only use bi_size. Secondly, when cloning a bio and reusing (a portion of) the original bio's biovec, in order to calculate bi_vcnt for the new bio we'd have to iterate over all the biovecs in the new bio - which is silly as it's not needed.

  So, don't use bi_vcnt anymore.

- The current interface allows the block layer to split bios as needed, so we could eliminate a lot of complexity particularly in stacked drivers. Code that creates bios can then create whatever size bios are convenient, and more importantly stacked drivers don't have to deal with both their own bio size limitations and the limitations of the underlying devices. Thus there's no need to define ->merge_bvec_fn() callbacks for individual block drivers.

3.3 Usage of helpers:

- The following helpers whose names have the suffix of *_all* can only be used on non-BIO_CLONED bio. They are usually used by filesystem code. Drivers shouldn't use them because the bio may have been split before it reached the driver.

```
bio_for_each_segment_all()
bio_for_each_bvec_all()
bio_first_bvec_all()
bio_first_page_all()
bio_last_bvec_all()
```

- The following helpers iterate over single-page segment. The passed 'struct bio_vec' will contain a single-page IO vector during the iteration:

```
bio_for_each_segment()
bio_for_each_segment_all()
```

- The following helpers iterate over multi-page bvec. The passed 'struct bio_vec' will contain a multi-page IO vector during the iteration:

```
bio_for_each_bvec()
bio_for_each_bvec_all()
rq_for_each_bvec()
```

CHAPTER

FOUR

MULTI-QUEUE BLOCK IO QUEUEING MECHANISM (BLK-MQ)

The Multi-Queue Block IO Queueing Mechanism is an API to enable fast storage devices to achieve a huge number of input/output operations per second (IOPS) through queueing and submitting IO requests to block devices simultaneously, benefiting from the parallelism offered by modern storage devices.

4.1 Introduction

4.1.1 Background

Magnetic hard disks have been the de facto standard from the beginning of the development of the kernel. The Block IO subsystem aimed to achieve the best performance possible for those devices with a high penalty when doing random access, and the bottleneck was the mechanical moving parts, a lot slower than any layer on the storage stack. One example of such optimization technique involves ordering read/write requests according to the current position of the hard disk head.

However, with the development of Solid State Drives and Non-Volatile Memories without mechanical parts nor random access penalty and capable of performing high parallel access, the bottleneck of the stack had moved from the storage device to the operating system. In order to take advantage of the parallelism in those devices' design, the multi-queue mechanism was introduced.

The former design had a single queue to store block IO requests with a single lock. That did not scale well in SMP systems due to dirty data in cache and the bottleneck of having a single lock for multiple processors. This setup also suffered with congestion when different processes (or the same process, moving to different CPUs) wanted to perform block IO. Instead of this, the blk-mq API spawns multiple queues with individual entry points local to the CPU, removing the need for a lock. A deeper explanation on how this works is covered in the following section (*Operation*).

4.1.2 Operation

When the userspace performs IO to a block device (reading or writing a file, for instance), blk-mq takes action: it will store and manage IO requests to the block device, acting as middleware between the userspace (and a file system, if present) and the block device driver.

blk-mq has two group of queues: software staging queues and hardware dispatch queues. When the request arrives at the block layer, it will try the shortest path possible: send it directly to the hardware queue. However, there are two cases that it might not do that: if there's an IO scheduler attached at the layer or if we want to try to merge requests. In both cases, requests will be sent to the software queue.

Then, after the requests are processed by software queues, they will be placed at the hardware queue, a second stage queue were the hardware has direct access to process those requests. However, if the hardware does not have enough resources to accept more requests, blk-mq will places requests on a temporary queue, to be sent in the future, when the hardware is able.

Software staging queues

The block IO subsystem adds requests in the software staging queues (represented by struct blk_mq_ctx) in case that they weren't sent directly to the driver. A request is one or more BIOs. They arrived at the block layer through the data structure struct bio. The block layer will then build a new structure from it, the struct request that will be used to communicate with the device driver. Each queue has its own lock and the number of queues is defined by a per-CPU or per-node basis.

The staging queue can be used to merge requests for adjacent sectors. For instance, requests for sector 3-6, 6-7, 7-9 can become one request for 3-9. Even if random access to SSDs and NVMs have the same time of response compared to sequential access, grouped requests for sequential access decreases the number of individual requests. This technique of merging requests is called plugging.

Along with that, the requests can be reordered to ensure fairness of system resources (e.g. to ensure that no application suffers from starvation) and/or to improve IO performance, by an IO scheduler.

IO Schedulers

There are several schedulers implemented by the block layer, each one following a heuristic to improve the IO performance. They are "pluggable" (as in plug and play), in the sense of they can be selected at run time using sysfs. You can read more about Linux's IO schedulers here. The scheduling happens only between requests in the same queue, so it is not possible to merge requests from different queues, otherwise there would be cache trashing and a need to have a lock for each queue. After the scheduling, the requests are eligible to be sent to the hardware. One of the possible schedulers to be selected is the NONE scheduler, the most straightforward one. It will just place requests on whatever software queue the process is running on, without any reordering. When the device starts processing

requests in the hardware queue (a.k.a. run the hardware queue), the software queues mapped to that hardware queue will be drained in sequence according to their mapping.

Hardware dispatch queues

The hardware queue (represented by *struct blk_mq_hw_ctx*) is a struct used by device drivers to map the device submission queues (or device DMA ring buffer), and are the last step of the block layer submission code before the low level device driver taking ownership of the request. To run this queue, the block layer removes requests from the associated software queues and tries to dispatch to the hardware.

If it's not possible to send the requests directly to hardware, they will be added to a linked list (`hctx->dispatch`) of requests. Then, next time the block layer runs a queue, it will send the requests laying at the `dispatch` list first, to ensure a fairness dispatch with those requests that were ready to be sent first. The number of hardware queues depends on the number of hardware contexts supported by the hardware and its device driver, but it will not be more than the number of cores of the system. There is no reordering at this stage, and each software queue has a set of hardware queues to send requests for.

Note: Neither the block layer nor the device protocols guarantee the order of completion of requests. This must be handled by higher layers, like the filesystem.

Tag-based completion

In order to indicate which request has been completed, every request is identified by an integer, ranging from 0 to the dispatch queue size. This tag is generated by the block layer and later reused by the device driver, removing the need to create a redundant identifier. When a request is completed in the drive, the tag is sent back to the block layer to notify it of the finalization. This removes the need to do a linear search to find out which IO has been completed.

4.1.3 Further reading

- Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems
- NOOP scheduler
- Null block device driver

4.2 Source code documentation

struct **blk_mq_hw_ctx**

State for a hardware queue facing the hardware block device

Definition

```
struct blk_mq_hw_ctx {
  struct {
    spinlock_t lock;
    struct list_head        dispatch;
    unsigned long           state;
  };
  struct delayed_work     run_work;
  cpumask_var_t cpumask;
  int next_cpu;
  int next_cpu_batch;
  unsigned long           flags;
  void *sched_data;
  struct request_queue    *queue;
  struct blk_flush_queue  *fq;
  void *driver_data;
  struct sbitmap          ctx_map;
  struct blk_mq_ctx       *dispatch_from;
  unsigned int            dispatch_busy;
  unsigned short          type;
  unsigned short          nr_ctx;
  struct blk_mq_ctx       **ctxs;
  spinlock_t dispatch_wait_lock;
  wait_queue_entry_t dispatch_wait;
  atomic_t wait_index;
  struct blk_mq_tags      *tags;
  struct blk_mq_tags      *sched_tags;
  unsigned long           queued;
  unsigned long           run;
#define BLK_MQ_MAX_DISPATCH_ORDER       7;
  unsigned long           dispatched[BLK_MQ_MAX_DISPATCH_ORDER];
  unsigned int            numa_node;
  unsigned int            queue_num;
  atomic_t nr_active;
  atomic_t elevator_queued;
  struct hlist_node       cpuhp_online;
  struct hlist_node       cpuhp_dead;
  struct kobject          kobj;
  unsigned long           poll_considered;
  unsigned long           poll_invoked;
  unsigned long           poll_success;
#ifdef CONFIG_BLK_DEBUG_FS;
  struct dentry           *debugfs_dir;
  struct dentry           *sched_debugfs_dir;
```

(continues on next page)

(continued from previous page)

```
#endif;
  struct list_head        hctx_list;
  struct srcu_struct      srcu[];
};
```

Members

{unnamed_struct}
anonymous

lock
Protects the dispatch list.

dispatch
Used for requests that are ready to be dispatched to the hardware but for some reason (e.g. lack of resources) could not be sent to the hardware. As soon as the driver can send new requests, requests at this list will be sent first for a fairer dispatch.

state
BLK_MQ_S_* flags. Defines the state of the hw queue (active, scheduled to restart, stopped).

run_work
Used for scheduling a hardware queue run at a later time.

cpumask
Map of available CPUs where this hctx can run.

next_cpu
Used by blk_mq_hctx_next_cpu() for round-robin CPU selection from **cpumask**.

next_cpu_batch
Counter of how many works left in the batch before changing to the next CPU.

flags
BLK_MQ_F_* flags. Defines the behaviour of the queue.

sched_data
Pointer owned by the IO scheduler attached to a request queue. It’s up to the IO scheduler how to use this pointer.

queue
Pointer to the request queue that owns this hardware context.

fq
Queue of requests that need to perform a flush operation.

driver_data
Pointer to data owned by the block driver that created this hctx

ctx_map
Bitmap for each software queue. If bit is on, there is a pending request in that software queue.

dispatch_from
Software queue to be used when no scheduler was selected.

dispatch_busy
Number used by blk_mq_update_dispatch_busy() to decide if the hw_queue is busy using Exponential Weighted Moving Average algorithm.

type
HCTX_TYPE_* flags. Type of hardware queue.

nr_ctx
Number of software queues.

ctxs
Array of software queues.

dispatch_wait_lock
Lock for dispatch_wait queue.

dispatch_wait
Waitqueue to put requests when there is no tag available at the moment, to wait for another try in the future.

wait_index
Index of next available dispatch_wait queue to insert requests.

tags
Tags owned by the block driver. A tag at this set is only assigned when a request is dispatched from a hardware queue.

sched_tags
Tags owned by I/O scheduler. If there is an I/O scheduler associated with a request queue, a tag is assigned when that request is allocated. Else, this member is not used.

queued
Number of queued requests.

run
Number of dispatched requests.

dispatched
Number of dispatch requests by queue.

numa_node
NUMA node the storage adapter has been connected to.

queue_num
Index of this hardware queue.

nr_active
Number of active requests. Only used when a tag set is shared across request queues.

elevator_queued
Number of queued requests on hctx.

cpuhp_online
List to store request if CPU is going to die

cpuhp_dead
List to store request if some CPU die.

kobj
Kernel object for sysfs.

poll_considered
Count times *blk_poll()* was called.

poll_invoked
Count how many requests *blk_poll()* polled.

poll_success
Count how many polled requests were completed.

debugfs_dir
debugfs directory for this hardware queue. Named as cpu<cpu_number>.

sched_debugfs_dir
debugfs directory for the scheduler.

hctx_list
if this hctx is not in use, this is an entry in q->unused_hctx_list.

srcu
Sleepable RCU. Use as lock when type of the hardware queue is blocking (BLK_MQ_F_BLOCKING). Must be the last member - see also blk_mq_hw_ctx_size().

struct **blk_mq_queue_map**
Map software queues to hardware queues

Definition

```
struct blk_mq_queue_map {
  unsigned int *mq_map;
  unsigned int nr_queues;
  unsigned int queue_offset;
};
```

Members

mq_map
CPU ID to hardware queue index map. This is an array with nr_cpu_ids elements. Each element has a value in the range [**queue_offset**, **queue_offset** + **nr_queues**).

nr_queues
Number of hardware queues to map CPU IDs onto.

queue_offset
First hardware queue to map onto. Used by the PCIe NVMe driver to map each hardware queue type (*enum hctx_type*) onto a distinct set of hardware queues.

enum **hctx_type**
Type of hardware queue

Constants

HCTX_TYPE_DEFAULT
All I/O not otherwise accounted for.

HCTX_TYPE_READ
Just for READ I/O.

HCTX_TYPE_POLL
Polled I/O of any kind.

HCTX_MAX_TYPES
Number of types of hctx.

struct **blk_mq_tag_set**
tag set that can be shared between request queues

Definition

```
struct blk_mq_tag_set {
  struct blk_mq_queue_map map[HCTX_MAX_TYPES];
  unsigned int            nr_maps;
  const struct blk_mq_ops *ops;
  unsigned int            nr_hw_queues;
  unsigned int            queue_depth;
  unsigned int            reserved_tags;
  unsigned int            cmd_size;
  int numa_node;
  unsigned int            timeout;
  unsigned int            flags;
  void *driver_data;
  atomic_t active_queues_shared_sbitmap;
  struct sbitmap_queue    __bitmap_tags;
  struct sbitmap_queue    __breserved_tags;
  struct blk_mq_tags      **tags;
  struct mutex            tag_list_lock;
  struct list_head        tag_list;
};
```

Members

map
One or more ctx -> hctx mappings. One map exists for each hardware queue type (*enum hctx_type*) that the driver wishes to support. There are no restrictions on maps being of the same size, and it's perfectly legal to share maps between types.

nr_maps
Number of elements in the **map** array. A number in the range [1, HCTX_MAX_TYPES].

ops
Pointers to functions that implement block driver behavior.

nr_hw_queues
Number of hardware queues supported by the block driver that owns this data structure.

queue_depth
Number of tags per hardware queue, reserved tags included.

reserved_tags
Number of tags to set aside for BLK_MQ_REQ_RESERVED tag allocations.

cmd_size
Number of additional bytes to allocate per request. The block driver owns these additional bytes.

numa_node
NUMA node the storage adapter has been connected to.

timeout
Request processing timeout in jiffies.

flags
Zero or more BLK_MQ_F_* flags.

driver_data
Pointer to data owned by the block driver that created this tag set.

active_queues_shared_sbitmap
number of active request queues per tag set.

__bitmap_tags
A shared tags sbitmap, used over all hctx's

__breserved_tags
A shared reserved tags sbitmap, used over all hctx's

tags
Tag sets. One tag set per hardware queue. Has **nr_hw_queues** elements.

tag_list_lock
Serializes tag_list accesses.

tag_list
List of the request queues that use this tag set. See also request_queue.tag_set_list.

struct **blk_mq_queue_data**
Data about a request inserted in a queue

Definition

```
struct blk_mq_queue_data {
  struct request *rq;
  bool last;
};
```

Members

rq
Request pointer.

last
If it is the last request in the queue.

struct **blk_mq_ops**

Callback functions that implements block driver behaviour.

Definition

```
struct blk_mq_ops {
  blk_status_t (*queue_rq)(struct blk_mq_hw_ctx *, const struct blk_
↪mq_queue_data *);
  void (*commit_rqs)(struct blk_mq_hw_ctx *);
  bool (*get_budget)(struct request_queue *);
  void (*put_budget)(struct request_queue *);
  enum blk_eh_timer_return (*timeout)(struct request *, bool);
  int (*poll)(struct blk_mq_hw_ctx *);
  void (*complete)(struct request *);
  int (*init_hctx)(struct blk_mq_hw_ctx *, void *, unsigned int);
  void (*exit_hctx)(struct blk_mq_hw_ctx *, unsigned int);
  int (*init_request)(struct blk_mq_tag_set *set, struct request *,␣
↪unsigned int, unsigned int);
  void (*exit_request)(struct blk_mq_tag_set *set, struct request *,
↪ unsigned int);
  void (*initialize_rq_fn)(struct request *rq);
  void (*cleanup_rq)(struct request *);
  bool (*busy)(struct request_queue *);
  int (*map_queues)(struct blk_mq_tag_set *set);
#ifdef CONFIG_BLK_DEBUG_FS;
  void (*show_rq)(struct seq_file *m, struct request *rq);
#endif;
};
```

Members

queue_rq
Queue a new request from block IO.

commit_rqs
If a driver uses bd->last to judge when to submit requests to hardware, it must define this function. In case of errors that make us stop issuing further requests, this hook serves the purpose of kicking the hardware (which the last request otherwise would have done).

get_budget
Reserve budget before queue request, once .queue_rq is run, it is driver' s responsibility to release the reserved budget. Also we have to handle failure case of .get_budget for avoiding I/O deadlock.

put_budget
Release the reserved budget.

timeout
Called on request timeout.

poll
Called to poll for completion of a specific tag.

complete

Mark the request as complete.

init_hctx
Called when the block layer side of a hardware queue has been set up, allowing the driver to allocate/init matching structures.

exit_hctx
Ditto for exit/teardown.

init_request
Called for every command allocated by the block layer to allow the driver to set up driver specific data.

Tag greater than or equal to queue_depth is for setting up flush request.

exit_request
Ditto for exit/teardown.

initialize_rq_fn
Called from inside blk_get_request().

cleanup_rq
Called before freeing one request which isn't completed yet, and usually for freeing the driver private data.

busy
If set, returns whether or not this queue currently is busy.

map_queues
This allows drivers specify their own queue mapping by overriding the setup-time function that builds the mq_map.

show_rq
Used by the debugfs implementation to show driver-specific information about a request.

enum mq_rq_state **blk_mq_rq_state**(struct request *rq)
read the current MQ_RQ_* state of a request

Parameters

struct request *rq
target request.

struct request ***blk_mq_rq_from_pdu**(void *pdu)
cast a PDU to a request

Parameters

void *pdu
the PDU (Protocol Data Unit) to be casted

Return

request

Description

Driver command data is immediately after the request. So subtract request size to get back to the original request.

void ***blk_mq_rq_to_pdu**(struct request *rq)
cast a request to a PDU

Parameters

struct request *rq
the request to be casted

Return

pointer to the PDU

Description

Driver command data is immediately after the request. So add request to get the PDU.

void **blk_mq_quiesce_queue**(struct request_queue *q)
wait until all ongoing dispatches have finished

Parameters

struct request_queue *q
request queue.

Note

this function does not prevent that the struct request end_io() callback function is invoked. Once this function is returned, we make sure no dispatch can happen until the queue is unquiesced via blk_mq_unquiesce_queue().

void **blk_mq_complete_request**(struct request *rq)
end I/O on a request

Parameters

struct request *rq
the request being processed

Description

Complete a request by scheduling the ->complete_rq operation.

void **blk_mq_start_request**(struct request *rq)
Start processing a request

Parameters

struct request *rq
Pointer to request to be started

Description

Function used by device drivers to notify the block layer that a request is going to be processed now, so blk layer can do proper initializations such as starting the timeout timer.

void **__blk_mq_run_hw_queue**(struct *blk_mq_hw_ctx* *hctx)
Run a hardware queue.

Parameters

struct blk_mq_hw_ctx *hctx
Pointer to the hardware queue to run.

Description

Send pending requests to the hardware.

void **__blk_mq_delay_run_hw_queue**(struct *blk_mq_hw_ctx* *hctx, bool async, unsigned long msecs)

Run (or schedule to run) a hardware queue.

Parameters

struct blk_mq_hw_ctx *hctx
Pointer to the hardware queue to run.

bool async
If we want to run the queue asynchronously.

unsigned long msecs
Microseconds of delay to wait before running the queue.

Description

If **!async**, try to run the queue now. Else, run the queue asynchronously and with a delay of **msecs**.

void **blk_mq_delay_run_hw_queue**(struct *blk_mq_hw_ctx* *hctx, unsigned long msecs)

Run a hardware queue asynchronously.

Parameters

struct blk_mq_hw_ctx *hctx
Pointer to the hardware queue to run.

unsigned long msecs
Microseconds of delay to wait before running the queue.

Description

Run a hardware queue asynchronously with a delay of **msecs**.

void **blk_mq_run_hw_queue**(struct *blk_mq_hw_ctx* *hctx, bool async)

Start to run a hardware queue.

Parameters

struct blk_mq_hw_ctx *hctx
Pointer to the hardware queue to run.

bool async
If we want to run the queue asynchronously.

Description

Check if the request queue is not in a quiesced state and if there are pending requests to be sent. If this is true, run the queue to send requests to hardware.

void **blk_mq_run_hw_queues**(struct request_queue *q, bool async)

Run all hardware queues in a request queue.

Parameters

struct request_queue *q
Pointer to the request queue to run.

bool async
If we want to run the queue asynchronously.

void **blk_mq_delay_run_hw_queues**(struct request_queue *q, unsigned long msecs)

Run all hardware queues asynchronously.

Parameters

struct request_queue *q
Pointer to the request queue to run.

unsigned long msecs
Microseconds of delay to wait before running the queues.

bool **blk_mq_queue_stopped**(struct request_queue *q)

check whether one or more hctxs have been stopped

Parameters

struct request_queue *q
request queue.

Description

The caller is responsible for serializing this function against blk_mq_{start,stop}_hw_queue().

void **blk_mq_request_bypass_insert**(struct request *rq, bool at_head, bool run_queue)

Insert a request at dispatch list.

Parameters

struct request *rq
Pointer to request to be inserted.

bool at_head
true if the request should be inserted at the head of the list.

bool run_queue
If we should run the hardware queue after inserting the request.

Description

Should only be used carefully, when the caller knows we want to bypass a potential IO scheduler on the target device.

void **blk_mq_try_issue_directly**(struct *blk_mq_hw_ctx* *hctx, struct request *rq, blk_qc_t *cookie)

Try to send a request directly to device driver.

Parameters

struct blk_mq_hw_ctx *hctx
Pointer of the associated hardware queue.

struct request *rq
Pointer to request to be sent.

blk_qc_t *cookie
Request queue cookie.

Description

If the device has enough resources to accept a new request now, send the request directly to device driver. Else, insert at hctx->dispatch queue, so we can try send it another time in the future. Requests inserted at this queue have higher priority.

blk_qc_t **blk_mq_submit_bio**(struct *bio* *bio)
Create and send a request to block device.

Parameters

struct bio *bio
Bio pointer.

Description

Builds up a request structure from **q** and **bio** and send to the device. The request may not be queued directly to hardware if: * This request can be merged with another one * We want to place request at plug queue for possible future merging * There is an IO scheduler active at this queue

It will not queue the request if there is an error with the bio, or at the request creation.

Return

Request queue cookie.

int **blk_poll**(struct request_queue *q, blk_qc_t cookie, bool spin)
poll for IO completions

Parameters

struct request_queue *q
the queue

blk_qc_t cookie
cookie passed back at IO submission time

bool spin
whether to spin for completions

Description

Poll for completions on the passed in queue. Returns number of completed entries found. If **spin** is true, then blk_poll will continue looping until at least one completion is found, unless the task is otherwise marked running (or we need to reschedule).

CHAPTER

FIVE

GENERIC BLOCK DEVICE CAPABILITY

This file documents the sysfs file `block/<disk>/capability`.

`capability` is a bitfield, printed in hexadecimal, indicating which capabilities a specific block device supports:

genhd capability flags

`GENHD_FL_REMOVABLE` (0x0001): indicates that the block device gives access to removable media. When set, the device remains present even when media is not inserted. Must not be set for devices which are removed entirely when the media is removed.

`GENHD_FL_CD` (0x0008): the block device is a CD-ROM-style device. Affects responses to the `CDROM_GET_CAPABILITY` ioctl.

`GENHD_FL_UP` (0x0010): indicates that the block device is "up", with a similar meaning to network interfaces.

`GENHD_FL_SUPPRESS_PARTITION_INFO` (0x0020): don't include partition information in `/proc/partitions` or in the output of printk_all_partitions(). Used for the null block device and some MMC devices.

`GENHD_FL_EXT_DEVT` (0x0040): the driver supports extended dynamic `dev_t`, i.e. it wants extended device numbers (`BLOCK_EXT_MAJOR`). This affects the maximum number of partitions.

`GENHD_FL_NATIVE_CAPACITY` (0x0080): based on information in the partition table, the device's capacity has been extended to its native capacity; i.e. the device has hidden capacity used by one of the partitions (this is a flag used so that native capacity is only ever unlocked once).

`GENHD_FL_BLOCK_EVENTS_ON_EXCL_WRITE` (0x0100): event polling is blocked whenever a writer holds an exclusive lock.

`GENHD_FL_NO_PART_SCAN` (0x0200): partition scanning is disabled. Used for loop devices in their default settings and some MMC devices.

`GENHD_FL_HIDDEN` (0x0400): the block device is hidden; it doesn't produce events, doesn't appear in sysfs, and doesn't have an associated `bdev`. Implies `GENHD_FL_SUPPRESS_PARTITION_INFO` and `GENHD_FL_NO_PART_SCAN`. Used for multipath devices.

CHAPTER

SIX

EMBEDDED DEVICE COMMAND LINE PARTITION PARSING

The “blkdevparts” command line option adds support for reading the block device partition table from the kernel command line.

It is typically used for fixed block (eMMC) embedded devices. It has no MBR, so saves storage space. Bootloader can be easily accessed by absolute address of data on the block device. Users can easily change the partition.

The format for the command line is just like mtdparts:

blkdevparts=<blkdev-def>[;<blkdev-def>]

<blkdev-def> := <blkdev-id>:<partdef>[,<partdef>]
<partdef> := <size>[@<offset>](part-name)

<blkdev-id>
block device disk name. Embedded device uses fixed block device. Its disk name is also fixed, such as: mmcblk0, mmcblk1, mmcblk0boot0.

<size>
partition size, in bytes, such as: 512, 1m, 1G. size may contain an optional suffix of (upper or lower case):

K, M, G, T, P, E.

“-” is used to denote all remaining space.

<offset>
partition start address, in bytes. offset may contain an optional suffix of (upper or lower case):

K, M, G, T, P, E.

(part-name)
partition name. Kernel sends uevent with “PARTNAME”. Application can create a link to block device partition with the name “PARTNAME”. User space application can access partition by partition name.

Example:

eMMC disk names are “mmcblk0” and “mmcblk0boot0”.

bootargs:

```
'blkdevparts=mmcblk0:1G(data0),1G(data1),-;
↪mmcblk0boot0:1m(boot),-(kernel)'
```

dmesg:

```
mmcblk0: p1(data0) p2(data1) p3()
mmcblk0boot0: p1(boot) p2(kernel)
```

CHAPTER

SEVEN

DATA INTEGRITY

7.1 1. Introduction

Modern filesystems feature checksumming of data and metadata to protect against data corruption. However, the detection of the corruption is done at read time which could potentially be months after the data was written. At that point the original data that the application tried to write is most likely lost.

The solution is to ensure that the disk is actually storing what the application meant it to. Recent additions to both the SCSI family protocols (SBC Data Integrity Field, SCC protection proposal) as well as SATA/T13 (External Path Protection) try to remedy this by adding support for appending integrity metadata to an I/O. The integrity metadata (or protection information in SCSI terminology) includes a checksum for each sector as well as an incrementing counter that ensures the individual sectors are written in the right order. And for some protection schemes also that the I/O is written to the right place on disk.

Current storage controllers and devices implement various protective measures, for instance checksumming and scrubbing. But these technologies are working in their own isolated domains or at best between adjacent nodes in the I/O path. The interesting thing about DIF and the other integrity extensions is that the protection format is well defined and every node in the I/O path can verify the integrity of the I/O and reject it if corruption is detected. This allows not only corruption prevention but also isolation of the point of failure.

7.2 2. The Data Integrity Extensions

As written, the protocol extensions only protect the path between controller and storage device. However, many controllers actually allow the operating system to interact with the integrity metadata (IMD). We have been working with several FC/SAS HBA vendors to enable the protection information to be transferred to and from their controllers.

The SCSI Data Integrity Field works by appending 8 bytes of protection information to each sector. The data + integrity metadata is stored in 520 byte sectors on disk. Data + IMD are interleaved when transferred between the controller and target. The T13 proposal is similar.

Because it is highly inconvenient for operating systems to deal with 520 (and 4104) byte sectors, we approached several HBA vendors and encouraged them to allow

separation of the data and integrity metadata scatter-gather lists.

The controller will interleave the buffers on write and split them on read. This means that Linux can DMA the data buffers to and from host memory without changes to the page cache.

Also, the 16-bit CRC checksum mandated by both the SCSI and SATA specs is somewhat heavy to compute in software. Benchmarks found that calculating this checksum had a significant impact on system performance for a number of workloads. Some controllers allow a lighter-weight checksum to be used when interfacing with the operating system. Emulex, for instance, supports the TCP/IP checksum instead. The IP checksum received from the OS is converted to the 16-bit CRC when writing and vice versa. This allows the integrity metadata to be generated by Linux or the application at very low cost (comparable to software RAID5).

The IP checksum is weaker than the CRC in terms of detecting bit errors. However, the strength is really in the separation of the data buffers and the integrity metadata. These two distinct buffers must match up for an I/O to complete.

The separation of the data and integrity metadata buffers as well as the choice in checksums is referred to as the Data Integrity Extensions. As these extensions are outside the scope of the protocol bodies (T10, T13), Oracle and its partners are trying to standardize them within the Storage Networking Industry Association.

7.3 3. Kernel Changes

The data integrity framework in Linux enables protection information to be pinned to I/Os and sent to/received from controllers that support it.

The advantage to the integrity extensions in SCSI and SATA is that they enable us to protect the entire path from application to storage device. However, at the same time this is also the biggest disadvantage. It means that the protection information must be in a format that can be understood by the disk.

Generally Linux/POSIX applications are agnostic to the intricacies of the storage devices they are accessing. The virtual filesystem switch and the block layer make things like hardware sector size and transport protocols completely transparent to the application.

However, this level of detail is required when preparing the protection information to send to a disk. Consequently, the very concept of an end-to-end protection scheme is a layering violation. It is completely unreasonable for an application to be aware whether it is accessing a SCSI or SATA disk.

The data integrity support implemented in Linux attempts to hide this from the application. As far as the application (and to some extent the kernel) is concerned, the integrity metadata is opaque information that's attached to the I/O.

The current implementation allows the block layer to automatically generate the protection information for any I/O. Eventually the intent is to move the integrity metadata calculation to userspace for user data. Metadata and other I/O that originates within the kernel will still use the automatic generation interface.

Some storage devices allow each hardware sector to be tagged with a 16-bit value. The owner of this tag space is the owner of the block device. I.e. the filesystem in

most cases. The filesystem can use this extra space to tag sectors as they see fit. Because the tag space is limited, the block interface allows tagging bigger chunks by way of interleaving. This way, 8*16 bits of information can be attached to a typical 4KB filesystem block.

This also means that applications such as fsck and mkfs will need access to manipulate the tags from user space. A passthrough interface for this is being worked on.

7.4 4. Block Layer Implementation Details

7.4.1 4.1 Bio

The data integrity patches add a new field to struct bio when CONFIG_BLK_DEV_INTEGRITY is enabled. bio_integrity(bio) returns a pointer to a struct bip which contains the bio integrity payload. Essentially a bip is a trimmed down struct bio which holds a bio_vec containing the integrity metadata and the required housekeeping information (bvec pool, vector count, etc.)

A kernel subsystem can enable data integrity protection on a bio by calling bio_integrity_alloc(bio). This will allocate and attach the bip to the bio.

Individual pages containing integrity metadata can subsequently be attached using bio_integrity_add_page().

bio_free() will automatically free the bip.

7.4.2 4.2 Block Device

Because the format of the protection data is tied to the physical disk, each block device has been extended with a block integrity profile (struct blk_integrity). This optional profile is registered with the block layer using blk_integrity_register().

The profile contains callback functions for generating and verifying the protection data, as well as getting and setting application tags. The profile also contains a few constants to aid in completing, merging and splitting the integrity metadata.

Layered block devices will need to pick a profile that's appropriate for all subdevices. blk_integrity_compare() can help with that. DM and MD linear, RAID0 and RAID1 are currently supported. RAID4/5/6 will require extra work due to the application tag.

7.5 5.0 Block Layer Integrity API

7.5.1 5.1 Normal Filesystem

The normal filesystem is unaware that the underlying block device is capable of sending/receiving integrity metadata. The IMD will be automatically generated by the block layer at submit_bio() time in case of a WRITE. A READ request will cause the I/O integrity to be verified upon completion.

IMD generation and verification can be toggled using the:

```
/sys/block/<bdev>/integrity/write_generate
```

and:

```
/sys/block/<bdev>/integrity/read_verify
```

flags.

7.5.2 5.2 Integrity-Aware Filesystem

A filesystem that is integrity-aware can prepare I/Os with IMD attached. It can also use the application tag space if this is supported by the block device.

bool bio_integrity_prep(bio);

To generate IMD for WRITE and to set up buffers for READ, the filesystem must call bio_integrity_prep(bio).

Prior to calling this function, the bio data direction and start sector must be set, and the bio should have all data pages added. It is up to the caller to ensure that the bio does not change while I/O is in progress. Complete bio with error if prepare failed for some reson.

7.5.3 5.3 Passing Existing Integrity Metadata

Filesystems that either generate their own integrity metadata or are capable of transferring IMD from user space can use the following calls:

*struct bip * bio_integrity_alloc(bio, gfp_mask, nr_pages);*

Allocates the bio integrity payload and hangs it off of the bio. nr_pages indicate how many pages of protection data need to be stored in the integrity bio_vec list (similar to bio_alloc()).

The integrity payload will be freed at bio_free() time.

int bio_integrity_add_page(bio, page, len, offset);

Attaches a page containing integrity metadata to an existing bio. The bio must have an existing bip, i.e. bio_integrity_alloc()

must have been called. For a WRITE, the integrity metadata in the pages must be in a format understood by the target device with the notable exception that the sector numbers will be remapped as the request traverses the I/O stack. This implies that the pages added using this call will be modified during I/O! The first reference tag in the integrity metadata must have a value of bip->bip_sector.

Pages can be added using bio_integrity_add_page() as long as there is room in the bip bio_vec array (nr_pages).

Upon completion of a READ operation, the attached pages will contain the integrity metadata received from the storage device. It is up to the receiver to process them and verify data integrity upon completion.

7.5.4 5.4 Registering A Block Device As Capable Of Exchanging Integrity Metadata

To enable integrity exchange on a block device the gendisk must be registered as capable:

int blk_integrity_register(gendisk, blk_integrity);

The blk_integrity struct is a template and should contain the following:

```
static struct blk_integrity my_profile = {
    .name                   = "STANDARDSBODY-TYPE-
↪VARIANT-CSUM",
    .generate_fn            = my_generate_fn,
    .verify_fn              = my_verify_fn,
    .tuple_size             = sizeof(struct my_tuple_
↪size),
    .tag_size               = <tag bytes per hw␣
↪sector>,
};
```

'name'is a text string which will be visible in sysfs. This is part of the userland API so chose it carefully and never change it. The format is standards body-type-variant. E.g. T10-DIF-TYPE1-IP or T13-EPP-0-CRC.

'generate_fn' generates appropriate integrity metadata (for WRITE).

'verify_fn' verifies that the data buffer matches the integrity metadata.

'tuple_size' must be set to match the size of the integrity metadata per sector. I.e. 8 for DIF and EPP.

'tag_size' must be set to identify how many bytes of tag space are available per hardware sector. For DIF this is either 2 or 0 depending on the value of the Control Mode Page ATO bit.

2007-12-24 Martin K. Petersen <martin.petersen@oracle.com>

CHAPTER

EIGHT

DEADLINE IO SCHEDULER TUNABLES

This little file attempts to document how the deadline io scheduler works. In particular, it will clarify the meaning of the exposed tunables that may be of interest to power users.

8.1 Selecting IO schedulers

Refer to *Switching Scheduler* for information on selecting an io scheduler on a per-device basis.

8.2 read_expire (in ms)

The goal of the deadline io scheduler is to attempt to guarantee a start service time for a request. As we focus mainly on read latencies, this is tunable. When a read request first enters the io scheduler, it is assigned a deadline that is the current time + the read_expire value in units of milliseconds.

8.3 write_expire (in ms)

Similar to read_expire mentioned above, but for writes.

8.4 fifo_batch (number of requests)

Requests are grouped into `batches` of a particular data direction (read or write) which are serviced in increasing sector order. To limit extra seeking, deadline expiries are only checked between batches. fifo_batch controls the maximum number of requests per batch.

This parameter tunes the balance between per-request latency and aggregate throughput. When low latency is the primary concern, smaller is better (where a value of 1 yields first-come first-served behaviour). Increasing fifo_batch generally improves throughput, at the cost of latency variation.

8.5 writes_starved (number of dispatches)

When we have to move requests from the io scheduler queue to the block device dispatch queue, we always give a preference to reads. However, we don't want to starve writes indefinitely either. So writes_starved controls how many times we give preference to reads over writes. When that has been done writes_starved number of times, we dispatch some writes based on the same criteria as reads.

8.6 front_merges (bool)

Sometimes it happens that a request enters the io scheduler that is contiguous with a request that is already on the queue. Either it fits in the back of that request, or it fits at the front. That is called either a back merge candidate or a front merge candidate. Due to the way files are typically laid out, back merges are much more common than front merges. For some work loads, you may even know that it is a waste of time to spend any time attempting to front merge requests. Setting front_merges to 0 disables this functionality. Front merges may still occur due to the cached last_merge hint, but since that comes at basically 0 cost we leave that on. We simply disable the rbtree front sector lookup when the io scheduler merge function is called.

Nov 11 2002, Jens Axboe <jens.axboe@oracle.com>

CHAPTER

NINE

INLINE ENCRYPTION

9.1 Background

Inline encryption hardware sits logically between memory and the disk, and can en/decrypt data as it goes in/out of the disk. Inline encryption hardware has a fixed number of "keyslots" - slots into which encryption contexts (i.e. the encryption key, encryption algorithm, data unit size) can be programmed by the kernel at any time. Each request sent to the disk can be tagged with the index of a keyslot (and also a data unit number to act as an encryption tweak), and the inline encryption hardware will en/decrypt the data in the request with the encryption context programmed into that keyslot. This is very different from full disk encryption solutions like self encrypting drives/TCG OPAL/ATA Security standards, since with inline encryption, any block on disk could be encrypted with any encryption context the kernel chooses.

9.2 Objective

We want to support inline encryption (IE) in the kernel. To allow for testing, we also want a crypto API fallback when actual IE hardware is absent. We also want IE to work with layered devices like dm and loopback (i.e. we want to be able to use the IE hardware of the underlying devices if present, or else fall back to crypto API en/decryption).

9.3 Constraints and notes

- IE hardware has a limited number of "keyslots" that can be programmed with an encryption context (key, algorithm, data unit size, etc.) at any time. One can specify a keyslot in a data request made to the device, and the device will en/decrypt the data using the encryption context programmed into that specified keyslot. When possible, we want to make multiple requests with the same encryption context share the same keyslot.
- We need a way for upper layers like filesystems to specify an encryption context to use for en/decrypting a struct bio, and a device driver (like UFS) needs to be able to use that encryption context when it processes the bio.
- We need a way for device drivers to expose their inline encryption capabilities in a unified way to the upper layers.

9.4 Design

We add a struct bio_crypt_ctx to struct bio that can represent an encryption context, because we need to be able to pass this encryption context from the upper layers (like the fs layer) to the device driver to act upon.

While IE hardware works on the notion of keyslots, the FS layer has no knowledge of keyslots - it simply wants to specify an encryption context to use while en/decrypting a bio.

We introduce a keyslot manager (KSM) that handles the translation from encryption contexts specified by the FS to keyslots on the IE hardware. This KSM also serves as the way IE hardware can expose its capabilities to upper layers. The generic mode of operation is: each device driver that wants to support IE will construct a KSM and set it up in its struct request_queue. Upper layers that want to use IE on this device can then use this KSM in the device's struct request_queue to translate an encryption context into a keyslot. The presence of the KSM in the request queue shall be used to mean that the device supports IE.

The KSM uses refcounts to track which keyslots are idle (either they have no encryption context programmed, or there are no in-flight struct bios referencing that keyslot). When a new encryption context needs a keyslot, it tries to find a keyslot that has already been programmed with the same encryption context, and if there is no such keyslot, it evicts the least recently used idle keyslot and programs the new encryption context into that one. If no idle keyslots are available, then the caller will sleep until there is at least one.

9.5 blk-mq changes, other block layer changes and blk-crypto-fallback

We add a pointer to a `bi_crypt_context` and `keyslot` to struct request. These will be referred to as the `crypto fields` for the request. This `keyslot` is the keyslot into which the `bi_crypt_context` has been programmed in the KSM of the `request_queue` that this request is being sent to.

We introduce `block/blk-crypto-fallback.c`, which allows upper layers to remain blissfully unaware of whether or not real inline encryption hardware is present underneath. When a bio is submitted with a target `request_queue` that doesn't support the encryption context specified with the bio, the block layer will en/decrypt the bio with the blk-crypto-fallback.

If the bio is a `WRITE` bio, a bounce bio is allocated, and the data in the bio is encrypted stored in the bounce bio - blk-mq will then proceed to process the bounce bio as if it were not encrypted at all (except when blk-integrity is concerned). `blk-crypto-fallback` sets the bounce bio's `bi_end_io` to an internal function that cleans up the bounce bio and ends the original bio.

If the bio is a `READ` bio, the bio's `bi_end_io` (and also `bi_private`) is saved and overwritten by `blk-crypto-fallback` to `bio_crypto_fallback_decrypt_bio`. The bio's `bi_crypt_context` is also overwritten with `NULL`, so that to the rest of the stack, the bio looks as if it was a regular bio that never had an encryption context specified. `bio_crypto_fallback_decrypt_bio` will decrypt the bio,

restore the original `bi_end_io` (and also `bi_private`) and end the bio again.

Regardless of whether real inline encryption hardware is used or the blk-crypto-fallback is used, the ciphertext written to disk (and hence the on-disk format of data) will be the same (assuming the hardware's implementation of the algorithm being used adheres to spec and functions correctly).

If a `request queue`'s inline encryption hardware claimed to support the encryption context specified with a bio, then it will not be handled by the `blk-crypto-fallback`. We will eventually reach a point in blk-mq when a struct request needs to be allocated for that bio. At that point, blk-mq tries to program the encryption context into the `request_queue`'s keyslot_manager, and obtain a keyslot, which it stores in its newly added `keyslot` field. This keyslot is released when the request is completed.

When the first bio is added to a request, `blk_crypto_rq_bio_prep` is called, which sets the request's `crypt_ctx` to a copy of the bio's `bi_crypt_context`. bio_crypt_do_front_merge is called whenever a subsequent bio is merged to the front of the request, which updates the `crypt_ctx` of the request so that it matches the newly merged bio's `bi_crypt_context`. In particular, the request keeps a copy of the `bi_crypt_context` of the first bio in its bio-list (blk-mq needs to be careful to maintain this invariant during bio and request merges).

To make it possible for inline encryption to work with request queue based layered devices, when a request is cloned, its `crypto fields` are cloned as well. When the cloned request is submitted, blk-mq programs the `bi_crypt_context` of the request into the clone's request_queue's keyslot manager, and stores the returned keyslot in the clone's `keyslot`.

9.6 API presented to users of the block layer

`struct blk_crypto_key` represents a crypto key (the raw key, size of the key, the crypto algorithm to use, the data unit size to use, and the number of bytes required to represent data unit numbers that will be specified with the `bi_crypt_context`).

`blk_crypto_init_key` allows upper layers to initialize such a `blk_crypto_key`.

`bio_crypt_set_ctx` should be called on any bio that a user of the block layer wants en/decrypted via inline encryption (or the blk-crypto-fallback, if hardware support isn't available for the desired crypto configuration). This function takes the `blk_crypto_key` and the data unit number (DUN) to use when en/decrypting the bio.

`blk_crypto_config_supported` allows upper layers to query whether or not the an encryption context passed to request queue can be handled by blk-crypto (either by real inline encryption hardware, or by the blk-crypto-fallback). This is useful e.g. when blk-crypto-fallback is disabled, and the upper layer wants to use an algorithm that may not supported by hardware - this function lets the upper layer know ahead of time that the algorithm isn't supported, and the upper layer can fallback to something else if appropriate.

`blk_crypto_start_using_key` - Upper layers must call this function on `blk_crypto_key` and a `request_queue` before using the key with any bio headed for that `request_queue`. This function ensures that either the hardware supports

the key’ s crypto settings, or the crypto API fallback has transforms for the needed mode allocated and ready to go. Note that this function may allocate an `skcipher`, and must not be called from the data path, since allocating `skciphers` from the data path can deadlock.

`blk_crypto_evict_key` *must* be called by upper layers before a `blk_crypto_key` is freed. Further, it *must* only be called only once there are no more in-flight requests that use that `blk_crypto_key`. `blk_crypto_evict_key` will ensure that a key is removed from any keyslots in inline encryption hardware that the key might have been programmed into (or the blk-crypto-fallback).

9.7 API presented to device drivers

A :c:type:`struct blk_keyslot_manager` should be set up by device drivers in the `request_queue` of the device. The device driver needs to call `blk_ksm_init` on the `blk_keyslot_manager`, which specifying the number of keyslots supported by the hardware.

The device driver also needs to tell the KSM how to actually manipulate the IE hardware in the device to do things like programming the crypto key into the IE hardware into a particular keyslot. All this is achieved through the struct blk_ksm_ll_ops field in the KSM that the device driver must fill up after initing the `blk_keyslot_manager`.

The KSM also handles runtime power management for the device when applicable (e.g. when it wants to program a crypto key into the IE hardware, the device must be runtime powered on) - so the device driver must also set the `dev` field in the ksm to point to the *struct device* for the KSM to use for runtime power management.

`blk_ksm_reprogram_all_keys` can be called by device drivers if the device needs each and every of its keyslots to be reprogrammed with the key it “should have” at the point in time when the function is called. This is useful e.g. if a device loses all its keys on runtime power down/up.

`blk_ksm_destroy` should be called to free up all resources used by a keyslot manager upon `blk_ksm_init`, once the `blk_keyslot_manager` is no longer needed.

9.8 Layered Devices

Request queue based layered devices like dm-rq that wish to support IE need to create their own keyslot manager for their request queue, and expose whatever functionality they choose. When a layered device wants to pass a clone of that request to another `request_queue`, blk-crypto will initialize and prepare the clone as necessary - see `blk_crypto_insert_cloned_request` in `blk-crypto.c`.

9.9 Future Optimizations for layered devices

Creating a keyslot manager for a layered device uses up memory for each keyslot, and in general, a layered device merely passes the request on to a "child" device, so the keyslots in the layered device itself are completely unused, and don't need any refcounting or keyslot programming. We can instead define a new type of KSM; the "passthrough KSM", that layered devices can use to advertise an unlimited number of keyslots, and support for any encryption algorithms they choose, while not actually using any memory for each keyslot. Another use case for the "passthrough KSM" is for IE devices that do not have a limited number of keyslots.

9.10 Interaction between inline encryption and blk integrity

At the time of this patch, there is no real hardware that supports both these features. However, these features do interact with each other, and it's not completely trivial to make them both work together properly. In particular, when a WRITE bio wants to use inline encryption on a device that supports both features, the bio will have an encryption context specified, after which its integrity information is calculated (using the plaintext data, since the encryption will happen while data is being written), and the data and integrity info is sent to the device. Obviously, the integrity info must be verified before the data is encrypted. After the data is encrypted, the device must not store the integrity info that it received with the plaintext data since that might reveal information about the plaintext data. As such, it must re-generate the integrity info from the ciphertext data and store that on disk instead. Another issue with storing the integrity info of the plaintext data is that it changes the on disk format depending on whether hardware inline encryption support is present or the kernel crypto API fallback is used (since if the fallback is used, the device will receive the integrity info of the ciphertext, not that of the plaintext).

Because there isn't any real hardware yet, it seems prudent to assume that hardware implementations might not implement both features together correctly, and disallow the combination for now. Whenever a device supports integrity, the kernel will pretend that the device does not support hardware inline encryption (by essentially setting the keyslot manager in the request_queue of the device to NULL). When the crypto API fallback is enabled, this means that all bios with and encryption context will use the fallback, and IO will complete as usual. When the fallback is disabled, a bio with an encryption context will be failed.

CHAPTER

TEN

BLOCK IO PRIORITIES

10.1 Intro

With the introduction of cfq v3 (aka cfq-ts or time sliced cfq), basic io priorities are supported for reads on files. This enables users to io nice processes or process groups, similar to what has been possible with cpu scheduling for ages. This document mainly details the current possibilities with cfq; other io schedulers do not support io priorities thus far.

10.2 Scheduling classes

CFQ implements three generic scheduling classes that determine how io is served for a process.

IOPRIO_CLASS_RT: This is the realtime io class. This scheduling class is given higher priority than any other in the system, processes from this class are given first access to the disk every time. Thus it needs to be used with some care, one io RT process can starve the entire system. Within the RT class, there are 8 levels of class data that determine exactly how much time this process needs the disk for on each service. In the future this might change to be more directly mappable to performance, by passing in a wanted data rate instead.

IOPRIO_CLASS_BE: This is the best-effort scheduling class, which is the default for any process that hasn't set a specific io priority. The class data determines how much io bandwidth the process will get, it's directly mappable to the cpu nice levels just more coarsely implemented. 0 is the highest BE prio level, 7 is the lowest. The mapping between cpu nice level and io nice level is determined as: io_nice = (cpu_nice + 20) / 5.

IOPRIO_CLASS_IDLE: This is the idle scheduling class, processes running at this level only get io time when no one else needs the disk. The idle class has no class data, since it doesn't really apply here.

10.3 Tools

See below for a sample ionice tool. Usage:

```
# ionice -c<class> -n<level> -p<pid>
```

If pid isn't given, the current process is assumed. IO priority settings are inherited on fork, so you can use ionice to start the process at a given level:

```
# ionice -c2 -n0 /bin/ls
```

will run ls at the best-effort scheduling class at the highest priority. For a running process, you can give the pid instead:

```
# ionice -c1 -n2 -p100
```

will change pid 100 to run at the realtime scheduling class, at priority 2.

ionice.c tool:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <getopt.h>
#include <unistd.h>
#include <sys/ptrace.h>
#include <asm/unistd.h>

extern int sys_ioprio_set(int, int, int);
extern int sys_ioprio_get(int, int);

#if defined(__i386__)
#define __NR_ioprio_set                 289
#define __NR_ioprio_get                 290
#elif defined(__ppc__)
#define __NR_ioprio_set                 273
#define __NR_ioprio_get                 274
#elif defined(__x86_64__)
#define __NR_ioprio_set                 251
#define __NR_ioprio_get                 252
#elif defined(__ia64__)
#define __NR_ioprio_set                 1274
#define __NR_ioprio_get                 1275
#else
#error "Unsupported arch"
#endif

static inline int ioprio_set(int which, int who, int ioprio)
{
      return syscall(__NR_ioprio_set, which, who, ioprio);
}
```

(continues on next page)

(continued from previous page)

```
static inline int ioprio_get(int which, int who)
{
      return syscall(__NR_ioprio_get, which, who);
}

enum {
      IOPRIO_CLASS_NONE,
      IOPRIO_CLASS_RT,
      IOPRIO_CLASS_BE,
      IOPRIO_CLASS_IDLE,
};

enum {
      IOPRIO_WHO_PROCESS = 1,
      IOPRIO_WHO_PGRP,
      IOPRIO_WHO_USER,
};

#define IOPRIO_CLASS_SHIFT    13

const char *to_prio[] = { "none", "realtime", "best-effort", "idle",
 ↪ };

int main(int argc, char *argv[])
{
      int ioprio = 4, set = 0, ioprio_class = IOPRIO_CLASS_BE;
      int c, pid = 0;

      while ((c = getopt(argc, argv, "+n:c:p:")) != EOF) {
              switch (c) {
              case 'n':
                      ioprio = strtol(optarg, NULL, 10);
                      set = 1;
                      break;
              case 'c':
                      ioprio_class = strtol(optarg, NULL, 10);
                      set = 1;
                      break;
              case 'p':
                      pid = strtol(optarg, NULL, 10);
                      break;
              }
      }

      switch (ioprio_class) {
              case IOPRIO_CLASS_NONE:
                      ioprio_class = IOPRIO_CLASS_BE;
                      break;
```

(continues on next page)

(continued from previous page)

```
                case IOPRIO_CLASS_RT:
                case IOPRIO_CLASS_BE:
                        break;
                case IOPRIO_CLASS_IDLE:
                        ioprio = 7;
                        break;
                default:
                        printf("bad prio class %d\n", ioprio_class);
                        return 1;
        }

        if (!set) {
                if (!pid && argv[optind])
                        pid = strtol(argv[optind], NULL, 10);

                ioprio = ioprio_get(IOPRIO_WHO_PROCESS, pid);

                printf("pid=%d, %d\n", pid, ioprio);

                if (ioprio == -1)
                        perror("ioprio_get");
                else {
                        ioprio_class = ioprio >> IOPRIO_CLASS_SHIFT;
                        ioprio = ioprio & 0xff;
                        printf("%s: prio %d\n", to_prio[ioprio_class],
↪ ioprio);
                }
        } else {
                if (ioprio_set(IOPRIO_WHO_PROCESS, pid, ioprio |␣
↪ioprio_class << IOPRIO_CLASS_SHIFT) == -1) {
                        perror("ioprio_set");
                        return 1;
                }

                if (argv[optind])
                        execvp(argv[optind], &argv[optind]);
        }

        return 0;
}
```

March 11 2005, Jens Axboe <jens.axboe@oracle.com>

CHAPTER

ELEVEN

KYBER I/O SCHEDULER TUNABLES

The only two tunables for the Kyber scheduler are the target latencies for reads and synchronous writes. Kyber will throttle requests in order to meet these target latencies.

11.1 read_lat_nsec

Target latency for reads (in nanoseconds).

11.2 write_lat_nsec

Target latency for synchronous writes (in nanoseconds).

CHAPTER

TWELVE

NULL BLOCK DEVICE DRIVER

12.1 Overview

The null block device (`/dev/nullb*`) is used for benchmarking the various block-layer implementations. It emulates a block device of X gigabytes in size. It does not execute any read/write operation, just mark them as complete in the request queue. The following instances are possible:

Multi-queue block-layer

- Request-based.
- Configurable submission queues per device.

No block-layer (Known as bio-based)

- Bio-based. IO requests are submitted directly to the device driver.
- Directly accepts bio data structure and returns them.

All of them have a completion queue for each core in the system.

12.2 Module parameters

queue_mode=[0-2]: Default: 2-Multi-queue
Selects which block-layer the module should instantiate with.

0	Bio-based
1	Single-queue (deprecated)
2	Multi-queue

home_node=[0–nr_nodes]: Default: NUMA_NO_NODE
Selects what CPU node the data structures are allocated from.

gb=[Size in GB]: Default: 250GB
The size of the device reported to the system.

bs=[Block size (in bytes)]: Default: 512 bytes
The block size reported to the system.

nr_devices=[Number of devices]: Default: 1
Number of block devices instantiated. They are instantiated as /dev/nullb0, etc.

irqmode=[0-2]: Default: 1-Soft-irq
The completion mode used for completing IOs to the block-layer.

0	None.
1	Soft-irq. Uses IPI to complete IOs across CPU nodes. Simulates the overhead when IOs are issued from another CPU node than the home the device is connected to.
2	Timer: Waits a specific period (completion_nsec) for each IO before completion.

completion_nsec=[ns]: Default: 10,000ns
Combined with irqmode=2 (timer). The time each completion event must wait.

submit_queues=[1..nr_cpus]: Default: 1
The number of submission queues attached to the device driver. If unset, it defaults to 1. For multi-queue, it is ignored when use_per_node_hctx module parameter is 1.

hw_queue_depth=[0..qdepth]: Default: 64
The hardware queue depth of the device.

12.2.1 Multi-queue specific parameters

use_per_node_hctx=[0/1]: Default: 0
Number of hardware context queues.

0	The number of submit queues are set to the value of the submit_queues parameter.
1	The multi-queue block layer is instantiated with a hardware dispatch queue for each CPU node in the system.

no_sched=[0/1]: Default: 0
Enable/disable the io scheduler.

0	nullb* use default blk-mq io scheduler
1	nullb* doesn’t use io scheduler

blocking=[0/1]: Default: 0
Blocking behavior of the request queue.

0	Register as a non-blocking blk-mq driver device.
1	Register as a blocking blk-mq driver device, null_blk will set the BLK_MQ_F_BLOCKING flag, indicating that it sometimes/always needs to block in its ->queue_rq() function.

shared_tags=[0/1]: Default: 0
Sharing tags between devices.

0	Tag set is not shared.
1	Tag set shared between devices for blk-mq. Only makes sense with nr_devices > 1, otherwise there's no tag set to share.

zoned=[0/1]: Default: 0
Device is a random-access or a zoned block device.

0	Block device is exposed as a random-access block device.
1	Block device is exposed as a host-managed zoned block device. Requires CONFIG_BLK_DEV_ZONED.

zone_size=[MB]: Default: 256
Per zone size when exposed as a zoned block device. Must be a power of two.

zone_nr_conv=[nr_conv]: Default: 0
The number of conventional zones to create when block device is zoned. If zone_nr_conv >= nr_zones, it will be reduced to nr_zones - 1.

CHAPTER

THIRTEEN

BLOCK LAYER SUPPORT FOR PERSISTENT RESERVATIONS

The Linux kernel supports a user space interface for simplified Persistent Reservations which map to block devices that support these (like SCSI). Persistent Reservations allow restricting access to block devices to specific initiators in a shared storage setup.

This document gives a general overview of the support ioctl commands. For a more detailed reference please refer to the SCSI Primary Commands standard, specifically the section on Reservations and the "PERSISTENT RESERVE IN" and "PERSISTENT RESERVE OUT" commands.

All implementations are expected to ensure the reservations survive a power loss and cover all connections in a multi path environment. These behaviors are optional in SPC but will be automatically applied by Linux.

13.1 The following types of reservations are supported:

- **PR_WRITE_EXCLUSIVE**
  Only the initiator that owns the reservation can write to the device. Any initiator can read from the device.
- **PR_EXCLUSIVE_ACCESS**
  Only the initiator that owns the reservation can access the device.
- **PR_WRITE_EXCLUSIVE_REG_ONLY**
  Only initiators with a registered key can write to the device, Any initiator can read from the device.
- **PR_EXCLUSIVE_ACCESS_REG_ONLY**
  Only initiators with a registered key can access the device.
- PR_WRITE_EXCLUSIVE_ALL_REGS

  Only initiators with a registered key can write to the device, Any initiator can read from the device. All initiators with a registered key are considered reservation holders. Please reference the SPC spec on the meaning of a reservation holder if you want to use this type.
- **PR_EXCLUSIVE_ACCESS_ALL_REGS**
  Only initiators with a registered key can access the device. All initiators

with a registered key are considered reservation holders. Please reference the SPC spec on the meaning of a reservation holder if you want to use this type.

13.2 The following ioctl are supported:

13.2.1 1. IOC_PR_REGISTER

This ioctl command registers a new reservation if the new_key argument is non-null. If no existing reservation exists old_key must be zero, if an existing reservation should be replaced old_key must contain the old reservation key.

If the new_key argument is 0 it unregisters the existing reservation passed in old_key.

13.2.2 2. IOC_PR_RESERVE

This ioctl command reserves the device and thus restricts access for other devices based on the type argument. The key argument must be the existing reservation key for the device as acquired by the IOC_PR_REGISTER, IOC_PR_REGISTER_IGNORE, IOC_PR_PREEMPT or IOC_PR_PREEMPT_ABORT commands.

13.2.3 3. IOC_PR_RELEASE

This ioctl command releases the reservation specified by key and flags and thus removes any access restriction implied by it.

13.2.4 4. IOC_PR_PREEMPT

This ioctl command releases the existing reservation referred to by old_key and replaces it with a new reservation of type for the reservation key new_key.

13.2.5 5. IOC_PR_PREEMPT_ABORT

This ioctl command works like IOC_PR_PREEMPT except that it also aborts any outstanding command sent over a connection identified by old_key.

13.2.6 6. IOC_PR_CLEAR

This ioctl command unregisters both key and any other reservation key registered with the device and drops any existing reservation.

13.3 Flags

All the ioctls have a flag field. Currently only one flag is supported:

- **PR_FL_IGNORE_KEY**
  Ignore the existing reservation key. This is commonly supported for IOC_PR_REGISTER, and some implementation may support the flag for IOC_PR_RESERVE.

For all unknown flags the kernel will return -EOPNOTSUPP.

CHAPTER

FOURTEEN

QUEUE SYSFS FILES

This text file will detail the queue files that are located in the sysfs tree for each block device. Note that stacked devices typically do not export any settings, since their queue merely functions are a remapping target. These files are the ones found in the /sys/block/xxx/queue/ directory.

Files denoted with a RO postfix are readonly and the RW postfix means read-write.

14.1 add_random (RW)

This file allows to turn off the disk entropy contribution. Default value of this file is ‘1’ (on).

14.2 chunk_sectors (RO)

This has different meaning depending on the type of the block device. For a RAID device (dm-raid), chunk_sectors indicates the size in 512B sectors of the RAID volume stripe segment. For a zoned block device, either host-aware or host-managed, chunk_sectors indicates the size in 512B sectors of the zones of the device, with the eventual exception of the last zone of the device which may be smaller.

14.3 dax (RO)

This file indicates whether the device supports Direct Access (DAX), used by CPU-addressable storage to bypass the pagecache. It shows ‘1’ if true, ‘0’ if not.

14.4 discard_granularity (RO)

This shows the size of internal allocation of the device in bytes, if reported by the device. A value of ‘0’ means device does not support the discard functionality.

14.5 discard_max_hw_bytes (RO)

Devices that support discard functionality may have internal limits on the number of bytes that can be trimmed or unmapped in a single operation. The discard_max_bytes parameter is set by the device driver to the maximum number of bytes that can be discarded in a single operation. Discard requests issued to the device must not exceed this limit. A discard_max_bytes value of 0 means that the device does not support discard functionality.

14.6 discard_max_bytes (RW)

While discard_max_hw_bytes is the hardware limit for the device, this setting is the software limit. Some devices exhibit large latencies when large discards are issued, setting this value lower will make Linux issue smaller discards and potentially help reduce latencies induced by large discard operations.

14.7 discard_zeroes_data (RO)

Obsolete. Always zero.

14.8 fua (RO)

Whether or not the block driver supports the FUA flag for write requests. FUA stands for Force Unit Access. If the FUA flag is set that means that write requests must bypass the volatile cache of the storage device.

14.9 hw_sector_size (RO)

This is the hardware sector size of the device, in bytes.

14.10 io_poll (RW)

When read, this file shows whether polling is enabled (1) or disabled (0). Writing '0' to this file will disable polling for this device. Writing any non-zero value will enable this feature.

14.11 io_poll_delay (RW)

If polling is enabled, this controls what kind of polling will be performed. It defaults to -1, which is classic polling. In this mode, the CPU will repeatedly ask for completions without giving up any time. If set to 0, a hybrid polling mode is used, where the kernel will attempt to make an educated guess at when the IO will complete. Based on this guess, the kernel will put the process issuing IO to sleep for an amount of time, before entering a classic poll loop. This mode might be a little slower than pure classic polling, but it will be more efficient. If set to a value larger than 0, the kernel will put the process issuing IO to sleep for this amount of microseconds before entering classic polling.

14.12 io_timeout (RW)

io_timeout is the request timeout in milliseconds. If a request does not complete in this time then the block driver timeout handler is invoked. That timeout handler can decide to retry the request, to fail it or to start a device recovery strategy.

14.13 iostats (RW)

This file is used to control (on/off) the iostats accounting of the disk.

14.14 logical_block_size (RO)

This is the logical block size of the device, in bytes.

14.15 max_discard_segments (RO)

The maximum number of DMA scatter/gather entries in a discard request.

14.16 max_hw_sectors_kb (RO)

This is the maximum number of kilobytes supported in a single data transfer.

14.17 max_integrity_segments (RO)

Maximum number of elements in a DMA scatter/gather list with integrity data that will be submitted by the block layer core to the associated block driver.

14.18 max_active_zones (RO)

For zoned block devices (zoned attribute indicating “host-managed” or “host-aware”), the sum of zones belonging to any of the zone states: EXPLICIT OPEN, IMPLICIT OPEN or CLOSED, is limited by this value. If this value is 0, there is no limit.

If the host attempts to exceed this limit, the driver should report this error with BLK_STS_ZONE_ACTIVE_RESOURCE, which user space may see as the EOVERFLOW errno.

14.19 max_open_zones (RO)

For zoned block devices (zoned attribute indicating “host-managed” or “host-aware”), the sum of zones belonging to any of the zone states: EXPLICIT OPEN or IMPLICIT OPEN, is limited by this value. If this value is 0, there is no limit.

If the host attempts to exceed this limit, the driver should report this error with BLK_STS_ZONE_OPEN_RESOURCE, which user space may see as the ETOOMANYREFS errno.

14.20 max_sectors_kb (RW)

This is the maximum number of kilobytes that the block layer will allow for a filesystem request. Must be smaller than or equal to the maximum size allowed by the hardware.

14.21 max_segments (RO)

Maximum number of elements in a DMA scatter/gather list that is submitted to the associated block driver.

14.22 max_segment_size (RO)

Maximum size in bytes of a single element in a DMA scatter/gather list.

14.23 minimum_io_size (RO)

This is the smallest preferred IO size reported by the device.

14.24 nomerges (RW)

This enables the user to disable the lookup logic involved with IO merging requests in the block layer. By default (0) all merges are enabled. When set to 1 only simple one-hit merges will be tried. When set to 2 no merge algorithms will be tried (including one-hit or more complex tree/hash lookups).

14.25 nr_requests (RW)

This controls how many requests may be allocated in the block layer for read or write requests. Note that the total allocated number may be twice this amount, since it applies only to reads or writes (not the accumulated sum).

To avoid priority inversion through request starvation, a request queue maintains a separate request pool per each cgroup when CONFIG_BLK_CGROUP is enabled, and this parameter applies to each such per-block-cgroup request pool. IOW, if there are N block cgroups, each request queue may have up to N request pools, each independently regulated by nr_requests.

14.26 nr_zones (RO)

For zoned block devices (zoned attribute indicating “host-managed” or “host-aware”), this indicates the total number of zones of the device. This is always 0 for regular block devices.

14.27 optimal_io_size (RO)

This is the optimal IO size reported by the device.

14.28 physical_block_size (RO)

This is the physical block size of device, in bytes.

14.29 read_ahead_kb (RW)

Maximum number of kilobytes to read-ahead for filesystems on this block device.

14.30 rotational (RW)

This file is used to stat if the device is of rotational type or non-rotational type.

14.31 rq_affinity (RW)

If this option is '1', the block layer will migrate request completions to the cpu "group" that originally submitted the request. For some workloads this provides a significant reduction in CPU cycles due to caching effects.

For storage configurations that need to maximize distribution of completion processing setting this option to '2' forces the completion to run on the requesting cpu (bypassing the "group" aggregation logic).

14.32 scheduler (RW)

When read, this file will display the current and available IO schedulers for this block device. The currently active IO scheduler will be enclosed in [] brackets. Writing an IO scheduler name to this file will switch control of this block device to that new IO scheduler. Note that writing an IO scheduler name to this file will attempt to load that IO scheduler module, if it isn't already present in the system.

14.33 write_cache (RW)

When read, this file will display whether the device has write back caching enabled or not. It will return "write back" for the former case, and "write through" for the latter. Writing to this file can change the kernels view of the device, but it doesn't alter the device state. This means that it might not be safe to toggle the setting from "write back" to "write through", since that will also eliminate cache flushes issued by the kernel.

14.34 write_same_max_bytes (RO)

This is the number of bytes the device can write in a single write-same command. A value of ‘0’ means write-same is not supported by this device.

14.35 wbt_lat_usec (RW)

If the device is registered for writeback throttling, then this file shows the target minimum read latency. If this latency is exceeded in a given window of time (see wb_window_usec), then the writeback throttling will start scaling back writes. Writing a value of ‘0’ to this file disables the feature. Writing a value of ‘-1’ to this file resets the value to the default setting.

14.36 throttle_sample_time (RW)

This is the time window that blk-throttle samples data, in millisecond. blk-throttle makes decision based on the samplings. Lower time means cgroups have more smooth throughput, but higher CPU overhead. This exists only when CONFIG_BLK_DEV_THROTTLING_LOW is enabled.

14.37 write_zeroes_max_bytes (RO)

For block drivers that support REQ_OP_WRITE_ZEROES, the maximum number of bytes that can be zeroed at once. The value 0 means that REQ_OP_WRITE_ZEROES is not supported.

14.38 zoned (RO)

This indicates if the device is a zoned block device and the zone model of the device if it is indeed zoned. The possible values indicated by zoned are “none” for regular block devices and “host-aware” or “host-managed” for zoned block devices. The characteristics of host-aware and host-managed zoned block devices are described in the ZBC (Zoned Block Commands) and ZAC (Zoned Device ATA Command Set) standards. These standards also define the “drive-managed” zone model. However, since drive-managed zoned block devices do not support zone commands, they will be treated as regular block devices and zoned will report “none”.

14.39 zone_write_granularity (RO)

This indicates the alignment constraint, in bytes, for write operations in sequential zones of zoned block devices (devices with a zoned attributed that reports “host-managed” or “host-aware”). This value is always 0 for regular block devices.

Jens Axboe <jens.axboe@oracle.com>, February 2009

CHAPTER

FIFTEEN

STRUCT REQUEST DOCUMENTATION

Jens Axboe <jens.axboe@oracle.com> 27/05/02

15.1 Short explanation of request members

Classification flags:

D	driver member
B	block layer member
I	I/O scheduler member

Unless an entry contains a D classification, a device driver must not access this member. Some members may contain D classifications, but should only be access through certain macros or functions (eg ->flags).

<linux/blkdev.h>

Member	Flag	Comment
struct list_head queuelist	BI	Organization on various internal queues
`void *elevator_private`	I	I/O scheduler private data
unsigned char cmd[16]	D	Driver can use this for setting up a cdb before execution, see blk_queue_prep_rq
unsigned long flags	DBI	Contains info about data direction, request type, etc.
int rq_status	D	Request status bits
kdev_t rq_dev	DBI	Target device
int errors	DB	Error counts
sector_t sector	DBI	Target location
unsigned long hard_nr_sectors	B	Used to keep sector sane
unsigned long nr_sectors	DBI	Total number of sectors in request
unsigned long hard_nr_sectors	B	Used to keep nr_sectors sane
unsigned short nr_phys_segments	DB	Number of physical scatter gather segments in a request
unsigned short nr_hw_segments	DB	Number of hardware scatter gather segments in a request
unsigned int current_nr_sectors	DB	Number of sectors in first segment of request
unsigned int hard_cur_sectors	B	Used to keep current_nr_sectors sane
int tag	DB	TCQ tag, if assigned
`void *special`	D	Free to be used by driver
`char *buffer`	D	Map of first segment, also see section on bouncing SECTION
`struct completion *waiting`	D	Can be used by driver to get signalled on request completion
`struct bio *bio`	DBI	First bio in request
`struct bio *biotail`	DBI	Last bio in request
`struct request_queue *q`	DB	Request queue this request belongs to
`struct request_list *rl`	B	Request list this request came from

CHAPTER

SIXTEEN

BLOCK LAYER STATISTICS IN /SYS/BLOCK/<DEV>/STAT

This file documents the contents of the /sys/block/<dev>/stat file.

The stat file provides several statistics about the state of block device <dev>.

Q. Why are there multiple statistics in a single file? Doesn't sysfs normally contain a single value per file?

A. By having a single file, the kernel can guarantee that the statistics represent a consistent snapshot of the state of the device. If the statistics were exported as multiple files containing one statistic each, it would be impossible to guarantee that a set of readings represent a single point in time.

The stat file consists of a single line of text containing 11 decimal values separated by whitespace. The fields are summarized in the following table, and described in more detail below.

Name	units	description
read I/Os	requests	number of read I/Os processed
read merges	requests	number of read I/Os merged with in-queue I/O
read sectors	sectors	number of sectors read
read ticks	milliseconds	total wait time for read requests
write I/Os	requests	number of write I/Os processed
write merges	requests	number of write I/Os merged with in-queue I/O
write sectors	sectors	number of sectors written
write ticks	milliseconds	total wait time for write requests
in_flight	requests	number of I/Os currently in flight
io_ticks	milliseconds	total time this block device has been active
time_in_queue	milliseconds	total wait time for all requests
discard I/Os	requests	number of discard I/Os processed
discard merges	requests	number of discard I/Os merged with in-queue I/O
discard sectors	sectors	number of sectors discarded
discard ticks	milliseconds	total wait time for discard requests
flush I/Os	requests	number of flush I/Os processed
flush ticks	milliseconds	total wait time for flush requests

16.1 read I/Os, write I/Os, discard I/0s

These values increment when an I/O request completes.

16.2 flush I/Os

These values increment when an flush I/O request completes.

Block layer combines flush requests and executes at most one at a time. This counts flush requests executed by disk. Not tracked for partitions.

16.3 read merges, write merges, discard merges

These values increment when an I/O request is merged with an already-queued I/O request.

16.4 read sectors, write sectors, discard_sectors

These values count the number of sectors read from, written to, or discarded from this block device. The "sectors" in question are the standard UNIX 512-byte sectors, not any device- or filesystem-specific block size. The counters are incremented when the I/O completes.

16.5 read ticks, write ticks, discard ticks, flush ticks

These values count the number of milliseconds that I/O requests have waited on this block device. If there are multiple I/O requests waiting, these values will increase at a rate greater than 1000/second; for example, if 60 read requests wait for an average of 30 ms, the read_ticks field will increase by 60*30 = 1800.

16.6 in_flight

This value counts the number of I/O requests that have been issued to the device driver but have not yet completed. It does not include I/O requests that are in the queue but not yet issued to the device driver.

16.7 io_ticks

This value counts the number of milliseconds during which the device has had I/O requests queued.

16.8 time_in_queue

This value counts the number of milliseconds that I/O requests have waited on this block device. If there are multiple I/O requests waiting, this value will increase as the product of the number of milliseconds times the number of requests waiting (see "read ticks" above for an example).

CHAPTER

SEVENTEEN

SWITCHING SCHEDULER

Each io queue has a set of io scheduler tunables associated with it. These tunables control how the io scheduler works. You can find these entries in:

```
/sys/block/<device>/queue/iosched
```

assuming that you have sysfs mounted on /sys. If you don't have sysfs mounted, you can do so by typing:

```
# mount none /sys -t sysfs
```

It is possible to change the IO scheduler for a given block device on the fly to select one of mq-deadline, none, bfq, or kyber schedulers - which can improve that device's throughput.

To set a specific scheduler, simply do this:

```
echo SCHEDNAME > /sys/block/DEV/queue/scheduler
```

where SCHEDNAME is the name of a defined IO scheduler, and DEV is the device name (hda, hdb, sga, or whatever you happen to have).

The list of defined schedulers can be found by simply doing a "cat /sys/block/DEV/queue/scheduler" - the list of valid names will be displayed, with the currently selected scheduler in brackets:

```
# cat /sys/block/sda/queue/scheduler
[mq-deadline] kyber bfq none
# echo none >/sys/block/sda/queue/scheduler
# cat /sys/block/sda/queue/scheduler
[none] mq-deadline kyber bfq
```

CHAPTER

EIGHTEEN

EXPLICIT VOLATILE WRITE BACK CACHE CONTROL

18.1 Introduction

Many storage devices, especially in the consumer market, come with volatile write back caches. That means the devices signal I/O completion to the operating system before data actually has hit the non-volatile storage. This behavior obviously speeds up various workloads, but it means the operating system needs to force data out to the non-volatile storage when it performs a data integrity operation like fsync, sync or an unmount.

The Linux block layer provides two simple mechanisms that let filesystems control the caching behavior of the storage device. These mechanisms are a forced cache flush, and the Force Unit Access (FUA) flag for requests.

18.2 Explicit cache flushes

The REQ_PREFLUSH flag can be OR ed into the r/w flags of a bio submitted from the filesystem and will make sure the volatile cache of the storage device has been flushed before the actual I/O operation is started. This explicitly guarantees that previously completed write requests are on non-volatile storage before the flagged bio starts. In addition the REQ_PREFLUSH flag can be set on an otherwise empty bio structure, which causes only an explicit cache flush without any dependent I/O. It is recommend to use the blkdev_issue_flush() helper for a pure cache flush.

18.3 Forced Unit Access

The REQ_FUA flag can be OR ed into the r/w flags of a bio submitted from the filesystem and will make sure that I/O completion for this request is only signaled after the data has been committed to non-volatile storage.

18.4 Implementation details for filesystems

Filesystems can simply set the REQ_PREFLUSH and REQ_FUA bits and do not have to worry if the underlying devices need any explicit cache flushing and how the Forced Unit Access is implemented. The REQ_PREFLUSH and REQ_FUA flags may both be set on a single bio.

18.5 Implementation details for bio based block drivers

These drivers will always see the REQ_PREFLUSH and REQ_FUA bits as they sit directly below the submit_bio interface. For remapping drivers the REQ_FUA bits need to be propagated to underlying devices, and a global flush needs to be implemented for bios with the REQ_PREFLUSH bit set. For real device drivers that do not have a volatile cache the REQ_PREFLUSH and REQ_FUA bits on non-empty bios can simply be ignored, and REQ_PREFLUSH requests without data can be completed successfully without doing any work. Drivers for devices with volatile caches need to implement the support for these flags themselves without any help from the block layer.

18.6 Implementation details for request_fn based block drivers

For devices that do not support volatile write caches there is no driver support required, the block layer completes empty REQ_PREFLUSH requests before entering the driver and strips off the REQ_PREFLUSH and REQ_FUA bits from requests that have a payload. For devices with volatile write caches the driver needs to tell the block layer that it supports flushing caches by doing:

```
blk_queue_write_cache(sdkp->disk->queue, true, false);
```

and handle empty REQ_OP_FLUSH requests in its prep_fn/request_fn. Note that REQ_PREFLUSH requests with a payload are automatically turned into a sequence of an empty REQ_OP_FLUSH request followed by the actual write by the block layer. For devices that also support the FUA bit the block layer needs to be told to pass through the REQ_FUA bit using:

```
blk_queue_write_cache(sdkp->disk->queue, true, true);
```

and the driver must handle write requests that have the REQ_FUA bit set in prep_fn/request_fn. If the FUA bit is not natively supported the block layer turns it into an empty REQ_OP_FLUSH request after the actual write.

INDEX