Linux Block Documentation

The kernel development community

Jan 15, 2023

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 250ms.

1.3.8 fifo_expire_sync

This parameter is used to set the timeout of synchronous requests. Default value of this is 125ms. 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

For each group, the following parameters can be set:

weight This specifies the default weight for the cgroup inside its parent. Available values: 1..1000 (default: 100).

For cgroup v1, it is set by writing the value to *blkio.bfq.weight*.

For cgroup v2, it is set by writing the value to *io.bfq.weight*. (with an optional prefix of *default* and a space).

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.

weight_device This specifies a per-device weight for the cgroup. The syntax is *minor:major weight*. A weight of *0* may be used to reset to the default weight.

For cgroup v1, it is set by writing the value to *blkio.bfq.weight_device*.

For cgroup v2, the file name is *io.bfq.weight*.

[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

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.

As of 5.12 bvec segments with zero bv_len are not supported.

2.1 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.

2.2 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.

2.3 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

THREE

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.

3.1 Introduction

3.1.1 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*).

3.1.2 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 where 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 driver, 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.

3.1.3 Further reading

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

3.2 Source code documentation

void **rq_list_move**(struct request **src, struct request **dst, struct request *rq, struct request *prev)
move a struct request from one list to another

Parameters

struct request **src The source list **rq** is currently in

struct request **dst The destination list that **rq** will be appended to

struct request *rq The request to move

struct request *prev The request preceding **rq** in **src** (NULL if **rq** is the head)

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;
  unsigned int            numa_node;
  unsigned int            queue_num;
  atomic_t nr_active;
  struct hlist_node       cpuhp_online;
  struct hlist_node       cpuhp_dead;
  struct kobject          kobj;
#ifdef CONFIG_BLK_DEBUG_FS;
  struct dentry           *debugfs_dir;
  struct dentry           *sched_debugfs_dir;
#endif;
  struct list_head        hctx_list;
};
```

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.

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.

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.

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.

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;
  struct blk_mq_tags      **tags;
  struct blk_mq_tags      *shared_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.

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

shared_tags

Shared set of tags. Has **nr_hw_queues** elements. If set, shared by all **tags**.

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 *);
  void (*queue_rqs)(struct request **rqlist);
  int (*get_budget)(struct request_queue *);
  void (*put_budget)(struct request_queue *, int);
  void (*set_rq_budget_token)(struct request *, int);
  int (*get_rq_budget_token)(struct request *);
  enum blk_eh_timer_return (*timeout)(struct request *, bool);
  int (*poll)(struct blk_mq_hw_ctx *, struct io_comp_batch *);
  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 (*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).

queue_rqs Queue a list of new requests. Driver is guaranteed that each request belongs to the same queue. If the driver doesn't empty the **rqlist** completely, then the rest will be queued individually by the block layer upon return.

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.

set_rq_budget_token store rq's budget token

get_rq_budget_token retrieve rq's budget token

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.

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_wait_quiesce_done**(struct request_queue *q)
wait until in-progress quiesce is done

Parameters

struct request_queue *q request queue.

Note

it is driver's responsibility for making sure that quiesce has been started.

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().

bool **blk_update_request**(struct request *req, blk_status_t error, unsigned int nr_bytes)
Complete multiple bytes without completing the request

Parameters

struct request *req the request being processed

blk_status_t error block status code

unsigned int nr_bytes number of bytes to complete for **req**

Description

Ends I/O on a number of bytes attached to **req**, but doesn't complete the request structure even if **req** doesn't have leftover. If **req** has leftover, sets it up for the next range of segments.

Passing the result of blk_rq_bytes() as **nr_bytes** guarantees `false` return from this function.

Note

The RQF_SPECIAL_PAYLOAD flag is ignored on purpose in this function except in the consistency check at the end of this function.

Return

`false` - this request doesn't have any more data `true` - this request has more data

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_execute_rq_nowait**(struct request *rq, bool at_head)
insert a request to I/O scheduler for execution

Parameters

struct request *rq request to insert

bool at_head insert request at head or tail of queue

Description

Insert a fully prepared request at the back of the I/O scheduler queue for execution. Don't wait for completion.

Note

This function will invoke **done** directly if the queue is dead.

blk_status_t **blk_execute_rq**(struct request *rq, bool at_head)
insert a request into queue for execution

Parameters

struct request *rq request to insert

bool at_head insert request at head or tail of queue

Description

Insert a fully prepared request at the back of the I/O scheduler queue for execution and wait for completion.

Return

The blk_status_t result provided to blk_mq_end_request().

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 Milliseconds 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 Milliseconds 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 Milliseconds 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)
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.

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.

void **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.

blk_status_t **blk_insert_cloned_request**(struct request *rq)
Helper for stacking drivers to submit a request

Parameters

struct request *rq the request being queued

void **blk_rq_unprep_clone**(struct request *rq)
Helper function to free all bios in a cloned request

Parameters

struct request *rq the clone request to be cleaned up

Description

Free all bios in **rq** for a cloned request.

int **blk_rq_prep_clone**(struct request *rq, struct request *rq_src, struct bio_set *bs, gfp_t gfp_mask, int (*bio_ctr)(struct bio*, struct bio*, void*), void *data)
Helper function to setup clone request

Parameters

struct request *rq the request to be setup

struct request *rq_src original request to be cloned

struct bio_set *bs bio_set that bios for clone are allocated from

gfp_t gfp_mask memory allocation mask for bio

int (*bio_ctr)(struct bio *, struct bio *, void *) setup function to be called for each clone bio. Returns `0` for success, non `0` for failure.

void *data private data to be passed to **bio_ctr**

Description

Clones bios in **rq_src** to **rq**, and copies attributes of **rq_src** to **rq**. Also, pages which the original bios are pointing to are not copied and the cloned bios just point same pages. So cloned bios must be completed before original bios, which means the caller must complete **rq** before **rq_src**.

void **blk_mq_destroy_queue**(struct request_queue *q)
shutdown a request queue

Parameters

struct request_queue *q request queue to shutdown

Description

This shuts down a request queue allocated by blk_mq_init_queue() and drops the initial reference. All future requests will failed with -ENODEV.

Context

can sleep

CHAPTER

FOUR

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

`GENHD_FL_HIDDEN`: the block device is hidden; it doesn't produce events, doesn't appear in sysfs, and can't be opened from userspace or using blkdev_get*. Used for the underlying components of multipath devices.

`GENHD_FL_NO_PART`: partition support is disabled. The kernel will not scan for partitions from add_disk, and users can't add partitions manually.

unsigned int **disk_openers**(struct gendisk *disk)
    returns how many openers are there for a disk

Parameters

struct gendisk *disk disk to check

Description

This returns the number of openers for a disk. Note that this value is only stable if disk->open_mutex is held.

Note

Due to a quirk in the block layer open code, each open partition is only counted once even if there are multiple openers.

blk_alloc_disk

`blk_alloc_disk (node_id)`

    allocate a gendisk structure

Parameters

node_id numa node to allocate on

Description

Allocate and pre-initialize a gendisk structure for use with BIO based drivers.

Context

can sleep

void **bio_end_io_acct**(struct *bio* *bio, unsigned long start_time)
end I/O accounting for bio based drivers

Parameters

struct bio *bio bio to end account for

unsigned long start_time start time returned by bio_start_io_acct()

CHAPTER

FIVE

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

SIX

DATA INTEGRITY

6.1 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.

6.2 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.

6.3 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.

6.4 4. Block Layer Implementation Details

6.4.1 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.

6.4.2 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.

6.5 5.0 Block Layer Integrity API

6.5.1 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.

6.5.2 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.

6.5.3 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.

6.5.4 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 user-land 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

SEVEN

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.

7.1 Selecting IO schedulers

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

7.2 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.

7.3 write_expire (in ms)

Similar to read_expire mentioned above, but for writes.

7.4 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.

7.5 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.

7.6 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

EIGHT

INLINE ENCRYPTION

8.1 Background

Inline encryption hardware sits logically between memory and disk, and can en/decrypt data as it goes in/out of the disk. For each I/O request, software can control exactly how the inline encryption hardware will en/decrypt the data in terms of key, algorithm, data unit size (the granularity of en/decryption), and data unit number (a value that determines the initialization vector(s)).

Some inline encryption hardware accepts all encryption parameters including raw keys directly in low-level I/O requests. However, most inline encryption hardware instead has a fixed number of "keyslots" and requires that the key, algorithm, and data unit size first be programmed into a keyslot. Each low-level I/O request then just contains a keyslot index and data unit number.

Note that inline encryption hardware is very different from traditional crypto accelerators, which are supported through the kernel crypto API. Traditional crypto accelerators operate on memory regions, whereas inline encryption hardware operates on I/O requests. Thus, inline encryption hardware needs to be managed by the block layer, not the kernel crypto API.

Inline encryption hardware is also very different from "self-encrypting drives", such as those based on the TCG Opal or ATA Security standards. Self-encrypting drives don't provide fine-grained control of encryption and provide no way to verify the correctness of the resulting ciphertext. Inline encryption hardware provides fine-grained control of encryption, including the choice of key and initialization vector for each sector, and can be tested for correctness.

8.2 Objective

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

8.3 Constraints and notes

- We need a way for upper layers (e.g. filesystems) to specify an encryption context to use for en/decrypting a bio, and device drivers (e.g. UFSHCD) need to be able to use that encryption context when they process the request. Encryption contexts also introduce constraints on bio merging; the block layer needs to be aware of these constraints.
- Different inline encryption hardware has different supported algorithms, supported data unit sizes, maximum data unit numbers, etc. We call these properties the "crypto capabilities". We need a way for device drivers to advertise crypto capabilities to upper layers in a generic way.
- Inline encryption hardware usually (but not always) requires that keys be programmed into keyslots before being used. Since programming keyslots may be slow and there may not be very many keyslots, we shouldn't just program the key for every I/O request, but rather keep track of which keys are in the keyslots and reuse an already-programmed keyslot when possible.
- Upper layers typically define a specific end-of-life for crypto keys, e.g. when an encrypted directory is locked or when a crypto mapping is torn down. At these times, keys are wiped from memory. We must provide a way for upper layers to also evict keys from any keyslots they are present in.
- When possible, device-mapper devices must be able to pass through the inline encryption support of their underlying devices. However, it doesn't make sense for device-mapper devices to have keyslots themselves.

8.4 Basic design

We introduce `struct blk_crypto_key` to represent an inline encryption key and how it will be used. This includes the actual bytes of the key; the size of the key; the algorithm and data unit size the key will be used with; and the number of bytes needed to represent the maximum data unit number the key will be used with.

We introduce `struct bio_crypt_ctx` to represent an encryption context. It contains a data unit number and a pointer to a blk_crypto_key. We add pointers to a bio_crypt_ctx to `struct bio` and `struct request`; this allows users of the block layer (e.g. filesystems) to provide an encryption context when creating a bio and have it be passed down the stack for processing by the block layer and device drivers. Note that the encryption context doesn't explicitly say whether to encrypt or decrypt, as that is implicit from the direction of the bio; WRITE means encrypt, and READ means decrypt.

We also introduce `struct blk_crypto_profile` to contain all generic inline encryption-related state for a particular inline encryption device. The blk_crypto_profile serves as the way that drivers for inline encryption hardware advertise their crypto capabilities and provide certain functions (e.g., functions to program and evict keys) to upper layers. Each device driver that wants to support inline encryption will construct a blk_crypto_profile, then associate it with the disk's request_queue.

The blk_crypto_profile also manages the hardware's keyslots, when applicable. This happens in the block layer, so that users of the block layer can just specify encryption contexts and don't need to know about keyslots at all, nor do device drivers need to care about most details of keyslot management.

Specifically, for each keyslot, the block layer (via the blk_crypto_profile) keeps track of which blk_crypto_key that keyslot contains (if any), and how many in-flight I/O requests are using it. When the block layer creates a `struct request` for a bio that has an encryption context, it grabs a keyslot that already contains the key if possible. Otherwise it waits for an idle keyslot (a keyslot that isn't in-use by any I/O), then programs the key into the least-recently-used idle keyslot using the function the device driver provided. In both cases, the resulting keyslot is stored in the `crypt_keyslot` field of the request, where it is then accessible to device drivers and is released after the request completes.

`struct request` also contains a pointer to the original bio_crypt_ctx. Requests can be built from multiple bios, and the block layer must take the encryption context into account when trying to merge bios and requests. For two bios/requests to be merged, they must have compatible encryption contexts: both unencrypted, or both encrypted with the same key and contiguous data unit numbers. Only the encryption context for the first bio in a request is retained, since the remaining bios have been verified to be merge-compatible with the first bio.

To make it possible for inline encryption to work with request_queue based layered devices, when a request is cloned, its encryption context is cloned as well. When the cloned request is submitted, it is then processed as usual; this includes getting a keyslot from the clone's target device if needed.

8.5 blk-crypto-fallback

It is desirable for the inline encryption support of upper layers (e.g. filesystems) to be testable without real inline encryption hardware, and likewise for the block layer's keyslot management logic. It is also desirable to allow upper layers to just always use inline encryption rather than have to implement encryption in multiple ways.

Therefore, we also introduce *blk-crypto-fallback*, which is an implementation of inline encryption using the kernel crypto API. blk-crypto-fallback is built into the block layer, so it works on any block device without any special setup. Essentially, when a bio with an encryption context is submitted to a request_queue that doesn't support that encryption context, the block layer will handle en/decryption of the bio using blk-crypto-fallback.

For encryption, the data cannot be encrypted in-place, as callers usually rely on it being unmodified. Instead, blk-crypto-fallback allocates bounce pages, fills a new bio with those bounce pages, encrypts the data into those bounce pages, and submits that "bounce" bio. When the bounce bio completes, blk-crypto-fallback completes the original bio. If the original bio is too large, multiple bounce bios may be required; see the code for details.

For decryption, blk-crypto-fallback "wraps" the bio's completion callback (`bi_complete`) and private data (`bi_private`) with its own, unsets the bio's encryption context, then submits the bio. If the read completes successfully, blk-crypto-fallback restores the bio's original completion callback and private data, then decrypts the bio's data in-place using the kernel crypto API. Decryption happens from a workqueue, as it may sleep. Afterwards, blk-crypto-fallback completes the bio.

In both cases, the bios that blk-crypto-fallback submits no longer have an encryption context. Therefore, lower layers only see standard unencrypted I/O.

blk-crypto-fallback also defines its own blk_crypto_profile and has its own "keyslots"; its keyslots contain `struct crypto_skcipher` objects. The reason for this is twofold. First, it allows the keyslot management logic to be tested without actual inline encryption hardware.

Second, similar to actual inline encryption hardware, the crypto API doesn't accept keys directly in requests but rather requires that keys be set ahead of time, and setting keys can be expensive; moreover, allocating a crypto_skcipher can't happen on the I/O path at all due to the locks it takes. Therefore, the concept of keyslots still makes sense for blk-crypto-fallback.

Note that regardless of whether real inline encryption hardware or blk-crypto-fallback is used, the ciphertext written to disk (and hence the on-disk format of data) will be the same (assuming that both the inline encryption hardware's implementation and the kernel crypto API's implementation of the algorithm being used adhere to spec and function correctly).

blk-crypto-fallback is optional and is controlled by the `CONFIG_BLK_INLINE_ENCRYPTION_FALLBACK` kernel configuration option.

8.6 API presented to users of the block layer

`blk_crypto_config_supported()` allows users to check ahead of time whether inline encryption with particular crypto settings will work on a particular request_queue – either via hardware or via blk-crypto-fallback. This function takes in a `struct blk_crypto_config` which is like blk_crypto_key, but omits the actual bytes of the key and instead just contains the algorithm, data unit size, etc. This function can be useful if blk-crypto-fallback is disabled.

`blk_crypto_init_key()` allows users to initialize a blk_crypto_key.

Users must call `blk_crypto_start_using_key()` before actually starting to use a blk_crypto_key on a request_queue (even if `blk_crypto_config_supported()` was called earlier). This is needed to initialize blk-crypto-fallback if it will be needed. This must not be called from the data path, as this may have to allocate resources, which may deadlock in that case.

Next, to attach an encryption context to a bio, users should call `bio_crypt_set_ctx()`. This function allocates a bio_crypt_ctx and attaches it to a bio, given the blk_crypto_key and the data unit number that will be used for en/decryption. Users don't need to worry about freeing the bio_crypt_ctx later, as that happens automatically when the bio is freed or reset.

Finally, when done using inline encryption with a blk_crypto_key on a request_queue, users must call `blk_crypto_evict_key()`. This ensures that the key is evicted from all keyslots it may be programmed into and unlinked from any kernel data structures it may be linked into.

In summary, for users of the block layer, the lifecycle of a blk_crypto_key is as follows:

1. `blk_crypto_config_supported()` (optional)
2. `blk_crypto_init_key()`
3. `blk_crypto_start_using_key()`
4. `bio_crypt_set_ctx()` (potentially many times)
5. `blk_crypto_evict_key()` (after all I/O has completed)
6. Zeroize the blk_crypto_key (this has no dedicated function)

If a blk_crypto_key is being used on multiple request_queues, then `blk_crypto_config_supported()` (if used), `blk_crypto_start_using_key()`, and `blk_crypto_evict_key()` must be called on each request_queue.

8.7 API presented to device drivers

A device driver that wants to support inline encryption must set up a blk_crypto_profile in the request_queue of its device. To do this, it first must call `blk_crypto_profile_init()` (or its resource-managed variant `devm_blk_crypto_profile_init()`), providing the number of keyslots.

Next, it must advertise its crypto capabilities by setting fields in the blk_crypto_profile, e.g. `modes_supported` and `max_dun_bytes_supported`.

It then must set function pointers in the `ll_ops` field of the blk_crypto_profile to tell upper layers how to control the inline encryption hardware, e.g. how to program and evict keyslots. Most drivers will need to implement `keyslot_program` and `keyslot_evict`. For details, see the comments for `struct blk_crypto_ll_ops`.

Once the driver registers a blk_crypto_profile with a request_queue, I/O requests the driver receives via that queue may have an encryption context. All encryption contexts will be compatible with the crypto capabilities declared in the blk_crypto_profile, so drivers don't need to worry about handling unsupported requests. Also, if a nonzero number of keyslots was declared in the blk_crypto_profile, then all I/O requests that have an encryption context will also have a keyslot which was already programmed with the appropriate key.

If the driver implements runtime suspend and its blk_crypto_ll_ops don't work while the device is runtime-suspended, then the driver must also set the `dev` field of the blk_crypto_profile to point to the `struct device` that will be resumed before any of the low-level operations are called.

If there are situations where the inline encryption hardware loses the contents of its keyslots, e.g. device resets, the driver must handle reprogramming the keyslots. To do this, the driver may call `blk_crypto_reprogram_all_keys()`.

Finally, if the driver used `blk_crypto_profile_init()` instead of `devm_blk_crypto_profile_init()`, then it is responsible for calling `blk_crypto_profile_destroy()` when the crypto profile is no longer needed.

8.8 Layered Devices

Request queue based layered devices like dm-rq that wish to support inline encryption need to create their own blk_crypto_profile 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()`.

8.9 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 setting the blk_crypto_profile 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

NINE

BLOCK IO PRIORITIES

9.1 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.

9.2 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.

9.3 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);
}

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;
              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

TEN

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.

10.1 read_lat_nsec

Target latency for reads (in nanoseconds).

10.2 write_lat_nsec

Target latency for synchronous writes (in nanoseconds).

CHAPTER

ELEVEN

NULL BLOCK DEVICE DRIVER

11.1 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.

11.2 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.

11.2.1 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

TWELVE

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.

12.1 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.

12.2 The following ioctl are supported:

12.2.1 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.

12.2.2 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.

12.2.3 3. IOC_PR_RELEASE

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

12.2.4 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.

12.2.5 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.

12.2.6 6. IOC_PR_CLEAR

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

12.3 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

THIRTEEN

STRUCT REQUEST DOCUMENTATION

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

13.1 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

FOURTEEN

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 17 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

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

These values increment when an I/O request completes.

14.2 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.

14.3 read merges, write merges, discard merges

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

14.4 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.

14.5 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.

14.6 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.

14.7 io_ticks

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

14.8 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

FIFTEEN

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

SIXTEEN

EXPLICIT VOLATILE WRITE BACK CACHE CONTROL

16.1 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.

16.2 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.

16.3 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.

16.4 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.

16.5 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.

16.6 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