# I/O Barriers

Tejun Heo <htejun@gmail.com>, July 22 2005

I/O barrier requests are used to guarantee ordering around the barrier requests. Unless you're crazy enough to use disk drives for implementing synchronization constructs (wow, sounds interesting...), the ordering is meaningful only for write requests for things like journal checkpoints. All requests queued before a barrier request must be finished (made it to the physical medium) before the barrier request is started, and all requests queued after the barrier request must be started only after the barrier request is finished (again, made it to the physical medium).

In other words, I/O barrier requests have the following two properties.

1. Request ordering

Requests cannot pass the barrier request. Preceding requests are processed before the barrier and following requests after.

Depending on what features a drive supports, this can be done in one of the following three ways.

i. For devices which have queue depth greater than 1 (TCQ devices) and support ordered tags, block layer can just issue the barrier as an ordered request and the lower level driver, controller and drive itself are responsible for making sure that the ordering constraint is met. Most modern SCSI controllers/drives should support this.

NOTE: SCSI ordered tag isn't currently used due to limitation in the SCSI midlayer, see the following random notes section.

ii. For devices which have queue depth greater than 1 but don't support ordered tags, block layer ensures that the requests preceding a barrier request finishes before issuing the barrier request. Also, it defers requests following the barrier until the barrier request is finished. Older SCSI controllers/drives and SATA drives fall in this category.

iii. Devices which have queue depth of 1. This is a degenerate case of ii. Just keeping issue order suffices. Ancient SCSI controllers/drives and IDE drives are in this category.

2. Forced flushing to physical medium

Again, if you're not gonna do synchronization with disk drives (dang, it sounds even more appealing now!), the reason you use I/O barriers is mainly to protect filesystem integrity when power failure or some other events abruptly stop the drive from operating and possibly make the drive lose data in its cache. So, I/O barriers need to guarantee that requests actually get written to non-volatile medium in order.

There are four cases,

i. No write-back cache. Keeping requests ordered is enough.

ii. Write-back cache but no flush operation. There's no way to guarantee physical-medium commit order. This kind of devices can't to I/O barriers.

iii. Write-back cache and flush operation but no FUA (forced unit access). We need two cache flushes - before and after the barrier request.

iv. Write-back cache, flush operation and FUA. We still need one flush to make sure requests preceding a barrier are written to medium, but post-barrier flush can be avoided by using FUA write on the barrier itself.

How to support barrier requests in drivers
---

All barrier handling is done inside block layer proper. All low level drivers have to are implementing its prepare_flush_fn and using one the following two functions to indicate what barrier type it supports and how to prepare flush requests. Note that the term 'ordered' is used to indicate the whole sequence of performing barrier requests including draining and flushing.

```
typedef void (prepare_flush_fn)(struct request_queue *q, struct request *rq);

int blk_queue_ordered(struct request_queue *q, unsigned ordered,
                      prepare_flush_fn *prepare_flush_fn);

@q                      : the queue in question
@ordered                : the ordered mode the driver/device supports
@prepare_flush_fn       : this function should prepare @rq such that it
                          flushes cache to physical medium when executed
```

For example, SCSI disk driver's prepare_flush_fn looks like the following.

```
static void sd_prepare_flush(struct request_queue *q, struct request *rq)
{
        memset(rq->cmd, 0, sizeof(rq->cmd));
        rq->cmd_type = REQ_TYPE_BLOCK_PC;
        rq->timeout = SD_TIMEOUT;
        rq->cmd[0] = SYNCHRONIZE_CACHE;
        rq->cmd_len = 10;
}
```

The following seven ordered modes are supported. The following table shows which mode should be used depending on what features a device/driver supports. In the leftmost column of table, QUEUE_ORDERED_ prefix is omitted from the mode names to save space.

The table is followed by description of each mode. Note that in the descriptions of QUEUE_ORDERED_DRAIN*, '=>' is used whereas '->' is used for QUEUE_ORDERED_TAG* descriptions. '=>' indicates that the preceding step must be complete before proceeding to the next step.

'->' indicates that the next step can start as soon as the previous step is issued.

| | write-back cache | ordered tag | flush | FUA |
|---|---|---|---|---|
| NONE | yes/no | N/A | no | N/A |
| DRAIN | no | no | N/A | N/A |
| DRAIN_FLUSH | yes | no | yes | no |
| DRAIN_FUA | yes | no | yes | yes |
| TAG | no | yes | N/A | N/A |
| TAG_FLUSH | yes | yes | yes | no |
| TAG_FUA | yes | yes | yes | yes |

QUEUE_ORDERED_NONE
I/O barriers are not needed and/or supported.

Sequence: N/A

QUEUE_ORDERED_DRAIN
Requests are ordered by draining the request queue and cache flushing isn't needed.

Sequence: drain => barrier

QUEUE_ORDERED_DRAIN_FLUSH
Requests are ordered by draining the request queue and both pre-barrier and post-barrier cache flushings are needed.

Sequence: drain => preflush => barrier => postflush

QUEUE_ORDERED_DRAIN_FUA
Requests are ordered by draining the request queue and pre-barrier cache flushing is needed. By using FUA on barrier request, post-barrier flushing can be skipped.

Sequence: drain => preflush => barrier

QUEUE_ORDERED_TAG
Requests are ordered by ordered tag and cache flushing isn't needed.

Sequence: barrier

QUEUE_ORDERED_TAG_FLUSH
Requests are ordered by ordered tag and both pre-barrier and post-barrier cache flushings are needed.

Sequence: preflush -> barrier -> postflush

QUEUE_ORDERED_TAG_FUA
Requests are ordered by ordered tag and pre-barrier cache flushing is needed. By using FUA on barrier request, post-barrier flushing can be skipped.

Sequence: preflush -> barrier

Random notes/caveats
---

* SCSI layer currently can't use TAG ordering even if the drive, controller and driver support it. The problem is that SCSI midlayer request dispatch function is not atomic. It releases queue lock and switch to SCSI host lock during issue and it's possible and likely to happen in time that requests change their relative positions. Once this problem is solved, TAG ordering can be enabled.

* Currently, no matter which ordered mode is used, there can be only one barrier request in progress. All I/O barriers are held off by block layer until the previous I/O barrier is complete. This doesn't make any difference for DRAIN ordered devices, but, for TAG ordered devices with very high command latency, passing multiple I/O barriers to low level *might* be helpful if they are very frequent. Well, this certainly is a non-issue. I'm writing this just to make clear that no two I/O barrier is ever passed to low-level driver.

* Completion order. Requests in ordered sequence are issued in order but not required to finish in order. Barrier implementation can handle out-of-order completion of ordered sequence. IOW, the requests MUST be processed in order but the hardware/software completion paths are allowed to reorder completion notifications - eg. current SCSI midlayer doesn't preserve completion order during error handling.

* Requeueing order. Low-level drivers are free to requeue any request after they removed it from the request queue with blkdev_dequeue_request(). As barrier sequence should be kept in order when requeued, generic elevator code takes care of putting requests in order around barrier. See blk_ordered_req_seq() and ELEVATOR_INSERT_REQUEUE handling in __elv_add_request() for details.

Note that block drivers must not requeue preceding requests while completing latter requests in an ordered sequence. Currently, no error checking is done against this.

* Error handling. Currently, block layer will report error to upper layer if any of requests in an ordered sequence fails. Unfortunately, this doesn't seem to be enough. Look at the following request flow. QUEUE_ORDERED_TAG_FLUSH is in use.

```
 [0] [1] [2] [3] [pre] [barrier] [post] < [4] [5] [6] ... >
                                          still in elevator
```

Let's say request [2], [3] are write requests to update file system metadata (journal or whatever) and [barrier] is used to mark that those updates are valid. Consider the following sequence.

 i. Requests [0] ~ [post] leaves the request queue and enters low-level driver.
 ii. After a while, unfortunately, something goes wrong and the drive fails [2]. Note that any of [0], [1] and [3] could have completed by this time, but [pre] couldn't have been finished

as the drive must process it in order and it failed before processing that command.

iii. Error handling kicks in and determines that the error is unrecoverable and fails [2], and resumes operation.

iv. [pre] [barrier] [post] gets processed.

v. *BOOM* power fails

The problem here is that the barrier request is *supposed* to indicate that filesystem update requests [2] and [3] made it safely to the physical medium and, if the machine crashes after the barrier is written, filesystem recovery code can depend on that. Sadly, that isn't true in this case anymore. IOW, the success of a I/O barrier should also be dependent on success of some of the preceding requests, where only upper layer (filesystem) knows what 'some' is.

This can be solved by implementing a way to tell the block layer which requests affect the success of the following barrier request and making lower lever drivers to resume operation on error only after block layer tells it to do so.

As the probability of this happening is very low and the drive should be faulty, implementing the fix is probably an overkill. But, still, it's there.

* In previous drafts of barrier implementation, there was fallback mechanism such that, if FUA or ordered TAG fails, less fancy ordered mode can be selected and the failed barrier request is retried automatically. The rationale for this feature was that as FUA is pretty new in ATA world and ordered tag was never used widely, there could be devices which report to support those features but choke when actually given such requests.

This was removed for two reasons 1. it's an overkill 2. it's impossible to implement properly when TAG ordering is used as low level drivers resume after an error automatically. If it's ever needed adding it back and modifying low level drivers accordingly shouldn't be difficult.