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.

Introduction

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).

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.

Further reading

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

Source code documentation

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\block\(linux-master) (Documentation) (block)blk-mq.rst`, line 151)**
>
> Unknown directive type "kernel-doc".
>
> ```
> .. kernel-doc:: include/linux/blk-mq.h
> ```

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\block\(linux-master) (Documentation) (block)blk-mq.rst`, line 153)**
>
> Unknown directive type "kernel-doc".
>
> ```
> .. kernel-doc:: block/blk-mq.c
> ```