An Introduction to the Linux Kernel Block I/O Stack

Based on Linux 5.11

Benjamin Block ‹bblock@de.ibm.com›

March 14th, 2021

IBM Deutschland Research & Development GmbH

Trademark Attribution Statement

Note:

Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.
IBM hardware products are manufactured Sync new parts, or new and serviceable used parts. Regardless, our warranty terms apply.
All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.
All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.
Information about non-IBM products is obtained Sync the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Outline

What is a Block Device?

Anatomy of a Block Device

I/O Flow in the Block Layer

What is a Block Device?

A Try at a Definition

In Linux, a *Block Device* is a hardware abstraction. It represents hardware whose **data is stored and accessed in fixed size blocks** of *n* bytes (e.g. 512, 2048, or 4096 bytes) [18].

In contrast to *Character Devices*, blocks on block devices can be **accessed in random-access pattern**, wherein the former only allows sequential access pattern [19].

Typically, and for this talk, block devices represent **persistent mass storage hardware**.

But not all block devices in Linux are backed by persistent storage (e.g. RAM Disks whose data is stored in memory), nor must all of them organize their data in fixed blocks (e.g. ECKD formatted DASDs whose data is stored in variable length records). Even so, they can be represented as such in Linux, because of the abstraction provided by the Kernel.

What is a ‘Block’ Anyway?

A *Block* is a fixed amount of bytes that is used in the communication with a block device and the associated hardware. But different layers in the software stack differ in the exact meaning and size:

- **Userspace Software:** application specific meaning; usually how much data is read from/written to files via a single system-call.
- **VFS:** unit of bytes in which I/O is done by file systems in Linux. Between 512, and `PAGE_SIZE` bytes (e.g. 4 KiB for x86 and s390x, may be as big as 1 MiB).
- **Hardware**: also referred to as *Sector*.
  - **Logical**: smallest unit in bytes that is *addressable* on the device.
  - **Physical**: smallest unit in bytes that the device can *operate* on without resorting to read-modify-write.
  - *Physical may be bigger than Logical block size*

Using Block Devices in Linux (Examples)

Listing available block devices:

```
# ls /sys/class/block
dasda  dasda1  dasda2  dm-0  dm-1  dm-2  dm-3  scma  sda  sdb
```

Reading from a block device:

```
# dd if=/dev/sdf of=/dev/null bs=2MiB
10737418240 bytes (11 GB, 10 GiB) copied
```

Listing the topology of a stacked block devices:

```
# lsblk -s /dev/mapper/rhel_t3545003-root
NAME                MAJ:MIN RM  SIZE RO TYPE  MOUNTPOINT
rhel_t3545003-root  253:11   0    9G  0 lvm   /
+-mpathf2           253:8    0    9G  0 part
  +-mpathf          253:5    0   10G  0 mpath
    +-sdf             8:80   0   10G  0 disk
    +-sdam           66:96   0   10G  0 disk
```

Some More Examples for Block Devices

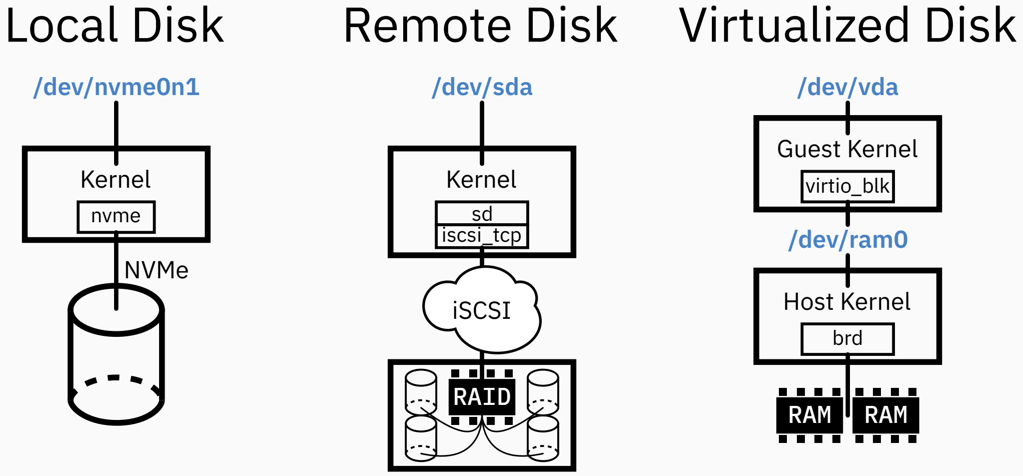


Figure 1: Examples for block device setups with different hardware backends [14].

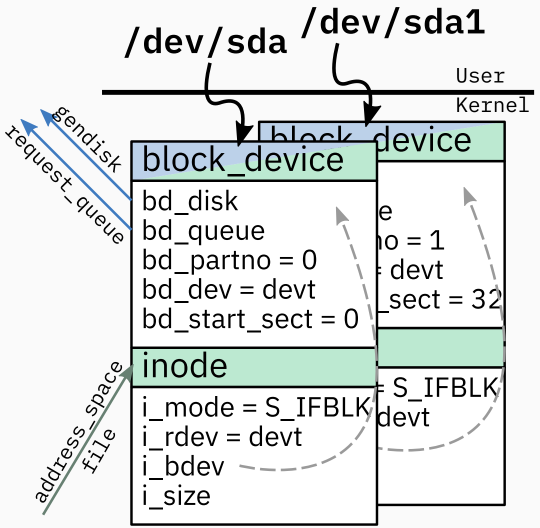
Anatomy of a Block Device

Structure of a Block Device: User Facing

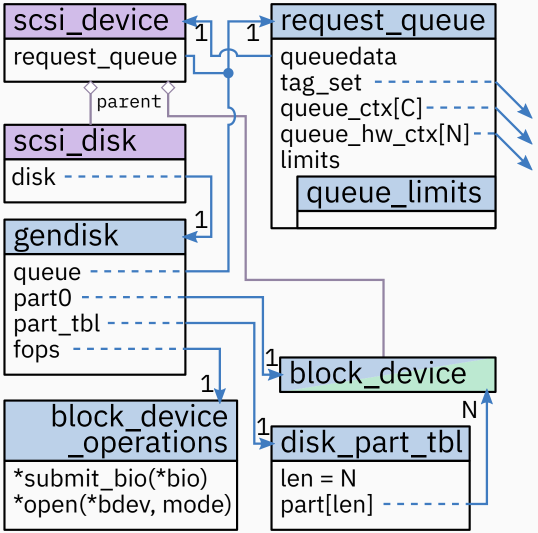
`block_device` Userspace interface; represents special file in `/dev`, and links to other kernel objects for the block device [4, 16]; partitions point to same disk and queue as whole device.

`inode` Each block device gets a virtual inode assigned, so it can be used in the VFS.

`file` and `address_space` Userspace processes open special file in `/dev`; in kernel represented as `file` with assigned `address_space` that point to block device inode.



Structure of a Block Device: Hardware Facing



gendisk and request_queue Central part of any block device; abstract hardware details for higher layers; **gendisk** represents the whole addressable space; **request_queue** how requests can be served

disk_part_tbl Points to partitions — represented as block_devices — backed by gendisk

scsi_device and scsi_disk Device drivers; provides common/mid layer for all SCSI-like hardware (incl. Serial ATA, SAS, iSCSI, FCP, …).

Queue Limits

- Attached to the Request Queue structure of a Block device
- Abstract hardware, firmware and device driver properties that influence how requests must be laid out
- Very important for stacked block devices
- For example:

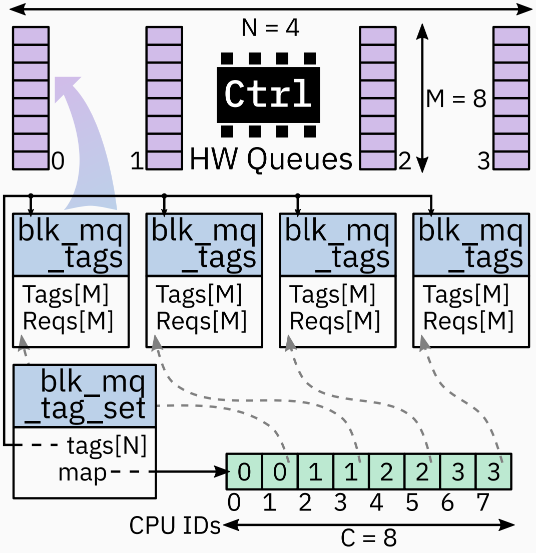
`logical_block_size`	Smallest possible unit in bytes that is *addressable* in a request.
`physical_block_size`	Smallest unit in bytes handled without read-modify-write.
`max_hw_sectors`	Amount of sectors (512 bytes) that a device can handle per request.
`io_opt`	Preferred size in bytes for requests to the device.
`max_sectors`	Softlimit used by VFS for buffered I/O (can be changed).
`max_segment_size`	Maximum size a segment in a request's scatter/gather list can have.
`max_segments`	Maximum amount of scatter/gather elements in a request.

Multi-Queue Request Queues

- In the past, request queues in Linux worked single threaded and without associating requests with particular processors
- Couldn’t properly exploit many-core systems and new storage hardware with more than one command queue (e.g.: NVMe); lots of cache thrashing
- Explicit *Multi-Queue* (MQ) support [3] was added with Linux 3.13: `blk-mq`
- I/O requests are scheduled on a hardware queue assigned to the I/O generating processor; responses are meant to be received on the same processor
- Structures necessary for I/O submission and response-handling are kept per processor; no shared state as much as possible
- With Linux 5.0 old single threaded queue implementation was removed

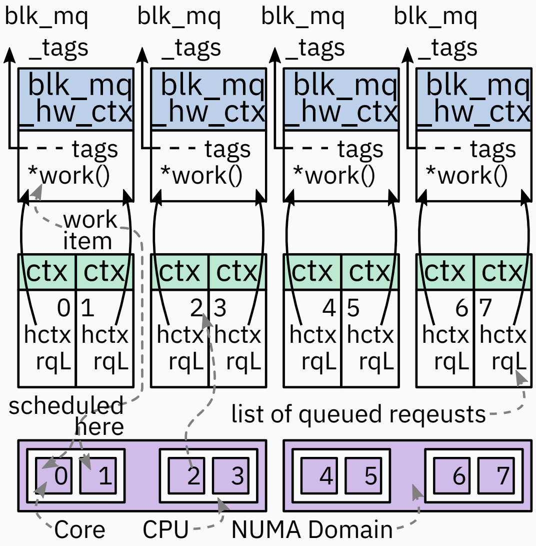
Block MQ Tag Set: Hardware Resource Allocation

- Per hardware queue resource allocation and management
- Requests (Reqs) are pre-allocated per HW queue (`blk_mq_tags`)
- Tags are index into the request array per queue, or per tag set (new in 5.10 [17])
- Allocation of tags handled via special data-structure: `sbitmap` [20]
- Tag set also provides mapping between CPU and hardware queue; objective is either 1 : 1 mapping, or cache proximity



Block MQ Soft- and Hardware-Context

- For a request queue (see pointers on page 7)
- Hardware context (`blk_mq_hw_ctx` = `hctx`) exists per hardware queue; hosts work item (`kblockd` work queue) scheduled on matching CPU; pulls requests out of associated `ctx` and submits them to hardware
- Software context (`blk_mq_ctx` = `ctx`) exists per CPU; queues requests in simple FIFO in absence of Elevator; associated with assigned HCTX as per tag set mapping



Block MQ Elevator / Scheduler

- `Elevator` = I/O Scheduler
- Can be set **optionally per request queue** (`/sys/class/block/<name>/queue/scheduler`)

`mq-deadline`	Forward port of old `deadline` scheduler; doesn't handle MQ context affinities; **default** for device with **1 hardware queue**; limits wait-time for requests to prevent starvation (500 ms for reads, 5 s for writes) [10, 18]
`kyber`	Only MQ native scheduler [10]; aims to meet certain latency targets (2 ms for reads, 10 ms for writes) by limiting the queue-depth dynamically
`bfq`	Only non-trivial I/O scheduler [6, 8, 9, 11] (replaces old CFQ scheduler); doesn't handle MQ context affinities; aims at providing fairness between I/O issuing processes
none	**Default** for device with **more than 1 hardware queue**; simply FIFO via MQ software context

What About Stacked Block Devices?

- Device-Mapper (dm) and Raid (md) use virtual/stacked block device on top of existing hardware-backed block devices ([15, 21])
  - Examples: RAID, LVM2, Multipathing
- Same structure as shown on page 6 and 7, without hardware specific structures, stacked on other `block_devices`
  - **BIO based**: doesn't have an Elevator, an own tag-set, nor any soft-, or hardware-contexts; modify I/O (BIO) after submission and immediately pass it on
  - **Request based**: have full set of infrastructure (only dm-multipath atm.); can queue requests; bypass lower-level queueing
- `queue_limits` of lower-level devices are aggregated into the "greatest common divisor", so that requests can be scheduled on any of them
- `holders/slaves` directories in sysfs show relationship

I/O Flow in the Block Layer

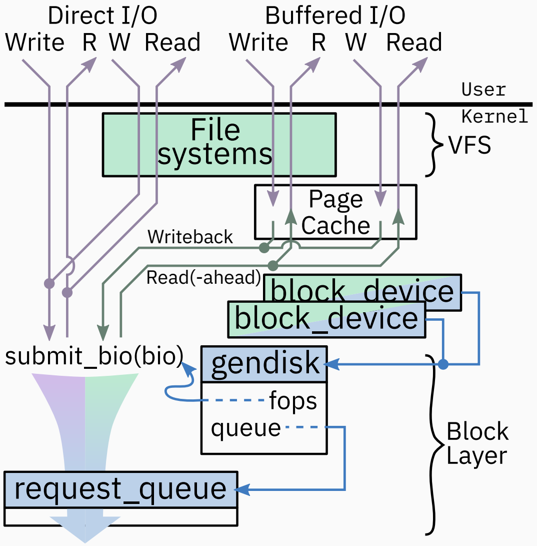
Submission of I/O Requests from Userspace (Simplified)

- I/O submission mainly categorized in 2 disciplines:

Buffered I/O Requests served via Page Cache; Writes cached and eventually — usually asynchronously — written to disk via *Writeback*; Reads served directly if fresh, otherwise read from disk synchronously

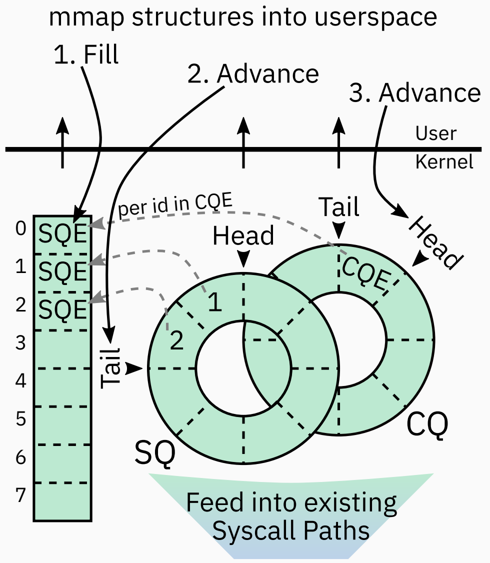
Direct I/O Requests served directly by backing disk; alignment and possibly size requirements; DMA directly into/from User memory possible

- For syncronous I/O system calls, tasks wait in state TASK_UNINTERRUPTIBLE

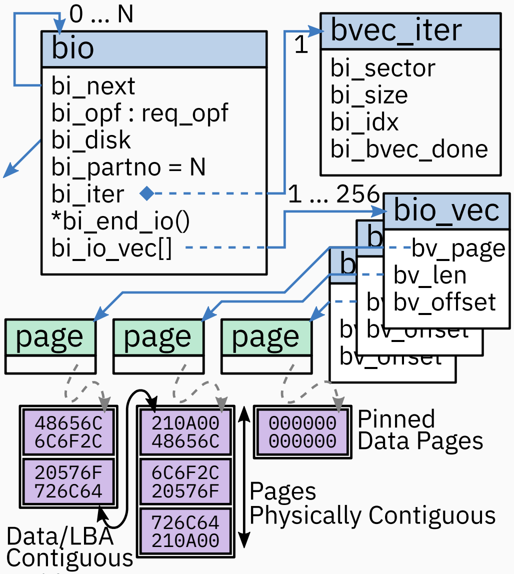


A New Asynchronous I/O Interface: io_uring

- With Linux 5.1 a new I/O submission API has been added: `io_uring` [12, 2]
- New set of System Calls to create set of ring structures: Submission Queue (SQ), Completion Queue (CQ), and Submission Queue Entries (SQE) array
- Structures shared between Kernel and User via `mmap(2)`
- Submission and Completion work asynchronously
- Utilizes standard syscall *backends* for calls like `readv(2)`, `writev(2)`, or `fsync(2)`; with same categories as on Page 14



The I/O Unit of the Block Layer: BIO



- BIOs represent in-flight I/O
- Application data kept separate; array of bio_vecs holds pointers to pages with application data (scatter/gather list)
- Position and progress managed in `bvec_iter`:

bi_sector	start sector
bi_size	size in bytes
bi_idx	current bvec index
bi_bvec_done	finished work in bytes

- BIOs might be split when queue limits (see page 8) exceeded; or cloned when same data goes to different places
- Data of single BIO limited to 4 GiB

Plugging and Merging

Plugging:

- When the VFS layer generates I/O requests and submits them for processing, it **plugs** the request queue of the target block device ([1])
- Requests generated while plug is active are not immediately submitted, but saved until **unplugging**
- **Unplugging** happens either explicitly, or during scheduled context switches

Merging:

- BIOs and requests are tried to be merged with already queued or *plugged* requests

  Back-Merging: The new data fits to the end of an existing request
  Front-Merging: The new data fits to the beginning of an existing request
- Merging is done by concatenating BIOs via `bi_next`
- Merges must not exceed queue limits

Entry Function into the Block Layer

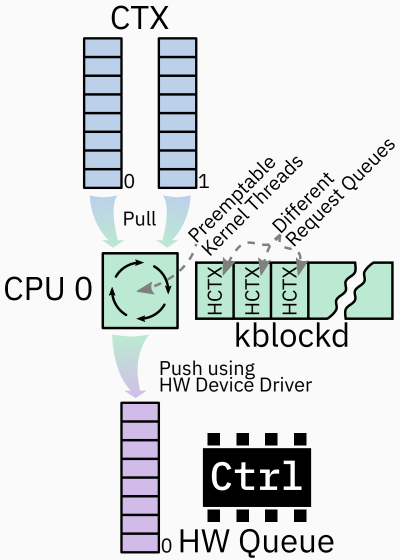
```
submit_bio(bio) {
  if (exists(current.bios)) {
    current.bios += bio
    return               thread local
  }
  On Stack: bios, old_bios
  current.bios ← bios
  do {
    old_bios ← current.bios
    current.bios = List()
    disk(bio)->submit_bio(bio)
    On Stack: same, lower
    lower = List()
    same = List()
    On Stack: q = queue(bio)
    while (bio ← pop(current.bios)) {
      if (q == queue(bio))
        same += bio
      else
        lower += bio
    }
    current.bios += lower
    current.bios += same
    current.bios += old_bios
  } while (bio = pop(current.bios))
  current.bios ← Null
  return
}
```

- When I/O is necessary, BIOs are generated and submitted into the block layer via `submit_bio()` ([4]).
- Doesn't guarantee synchronous processing; callback via `bio->bi_end_io(bio)`.
- One submitted BIO can turn into several more (block queue limits, stacked device, ...); each is also submitted via `submit_bio()`.
- Especially for stacked devices this could exhaust kernel stack space → turn recursion into iteration (approx.: depth-first search with stack)

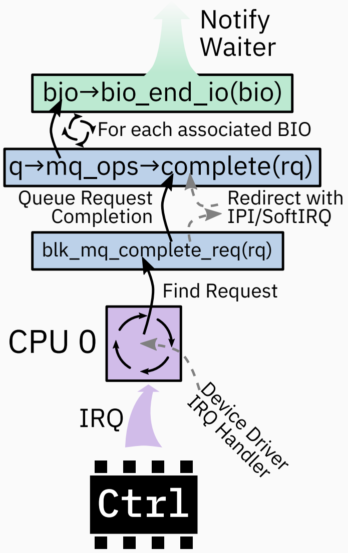
← Pseudo-code representation of functionality

Request Submission and Dispatch

- Once a BIO reaches a request queue (see Page 12 and 11) the submitting task tries to get Tag from the associated HCTX
  - Creates back pressure if not possible
- The BIO is added to the associated `Request`; via linking, this could be multiple BIOs per request
- The `Request` is inserted into software context FIFO queue, or Elevator (if enabled); the HCTX work-item is queued into `kblockd`
- Associated CPU executes HCTX work-item
- Work-item pulls queued requests out of associated software contexts or Elevators and hands them to HW device driver



Request Completion



- Once a request is processed, device drivers usually get notified via an interrupt
- To keep data CPU local, interrupts should be bound to CPU of associated MQ software context (platform-/controller-/driver-dependant)
  - `blk-mq` resorts to IPI or SoftIRQ otherwise
- The device driver is responsible for determining the corresponding block layer request for the signaled completion
- Progress is measured by how many bytes were successfully completed; might cause re-schedule
- Process of completing a request is a bunch of **callbacks** to notify the waiting user-/kernel-thread

Block Layer Polling

- Similar to high speed networking, with **high speed storage targets** it can be beneficial to use **Polling instead of Waiting for Interrupts** to handle request completion
  - Decreases response times and reduces overhead produced by interrupts on fast devices
- **Available in Linux since 4.10** [7]; only supported by NVMe at this point (support for SCSI merged for 5.13 [13], support for dm in work [26])
  - Enable per request queue: **echo** 1 > /sys/class/block/<name>/queue/io_poll
  - Enable for NVMe with module parameter: nvme.poll_queues=N
- Device driver creates **separate HW queues** that have interrupts disabled
- Whether polling is used is controlled by applications (only with **Direct I/O** currently):
  - Pass **`RWF_HIPRI`** to **`readv(2)`**/**`writev(2)`**
  - Pass **`IORING_SETUP_IOPOLL`** to **`io_uring_setup(2)`** for `io_uring`
- When used, application threads that issued I/O, or `io_uring` worker threads, actively poll in HW queues whether the issued request has been completed

Closing

Summary

- Block devices are a **hardware abstraction** to allow uniform access to a range of diverse hardware
- The entry into the block layer is provided by **`block_device`** device-nodes, which are backed by a **`gendisk`** — representing the block-addressable storage space — , and a **`request_queue`** — providing a generic way to queue requests against the hardware
- Special care is taken to allow **processor local processing** of I/O requests and responses
- Userspace requests mainly categorized in **`Buffered`** and **`Direct I/O`**
- Central structure for transporting information about in-flight I/O is the **BIO**; it allows for cloning, splitting and merging without copying payload
- Processing of **I/O is fundamentally asynchronous** in the kernel, requests happen in a different context than responses, and are only synchronized via wait/notify mechanisms

Made with LaTeX and Inkscape ♥

Questions?

IBM Deutschland Research & Development

Headquarters Böblingen

- Big parts of the support for Linux on IBM Z — Kernel and Userland — are done at the IBM Laboratory in Böblingen
- We follow a strict upstream policy and do not — with scarce exceptions — ship code that is not accepted in the respective upstream project
- Parts of the hard- and firmware for the IBM Mainframes are also done in Böblingen
- `https://www.ibm.com/de-de/marketing/entwicklung/about.html`

Backup Slides

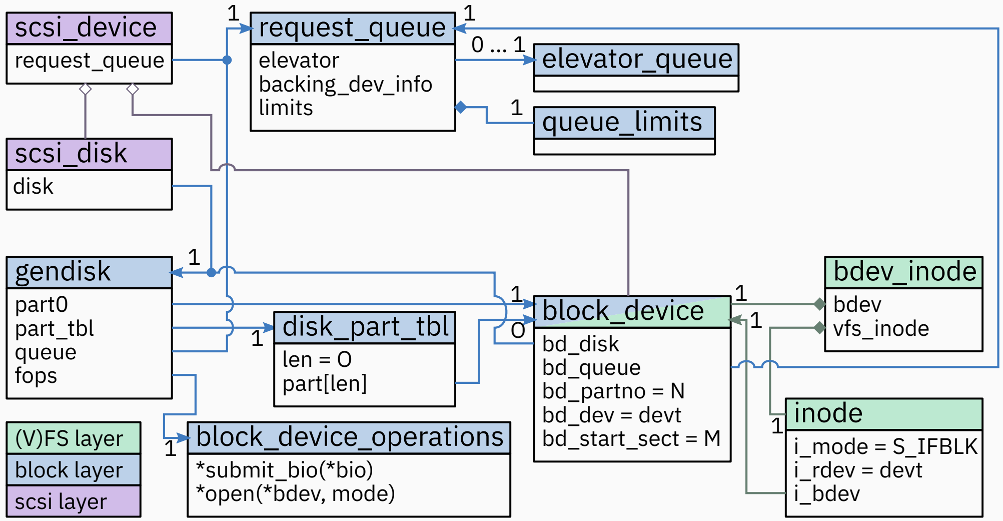
But What About Zoned Storage?

- Introduced with the advent of SMR Disks, but now also in NVMe specification
- Tracks on the disk overlap; overwriting a single track means overwriting a bunch of other tracks as well [24]

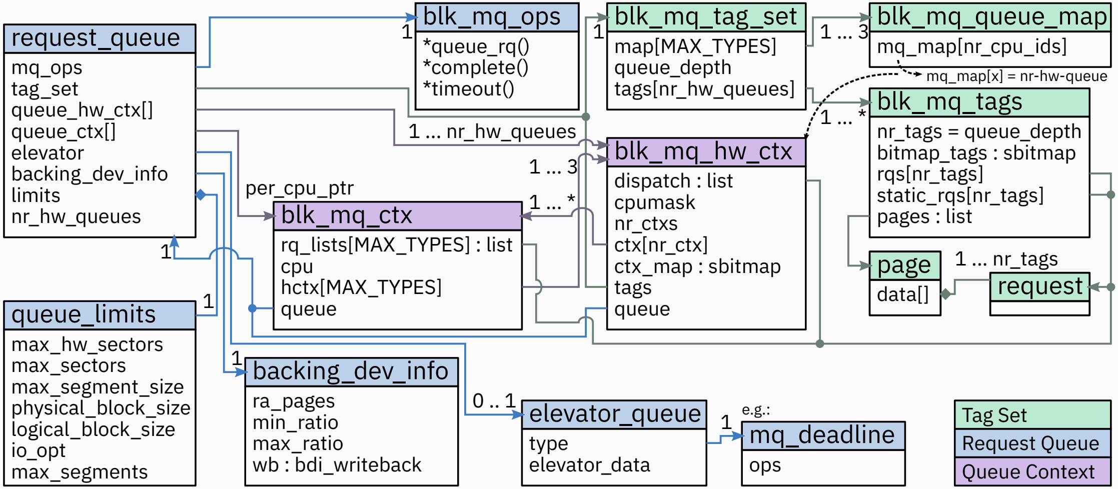
→ Data is organized in “bands” of tracks: **zones**. Each zone only **written sequentially**; **out-of-order writes** only **after reset** of whole zone. **Random read access** remains possible.

- Supported by the Linux block layer, but breaks with previous definition
- Direct use only via aid by special ioctls [23], via special device-mapper target [22], or via special file system [25]
- **Gonna ignore this for the rest of the talk**

Structure of a Block Device: Whole Picture



Structure of a MQ Request Queue: Whole Picture



Glossary i

BIO	BIO: represents metadata and data for I/O in the Linux block layer; no hardware specific information.
CFQ	Completely Fair Queuing: deprecated I/O scheduler for single queue block layer.
DASD	Direct-Access Storage Device: disk storage type used by IBM Z via FICON.
dm	Device-Mapper: low level volume manager; allows to specify mappings for ranges of logical sectors; higher level volume managers such as LVM2 use this driver.
DMA	Direct Memory Access: hardware components can access main memory without CPU involvement.
ECKD	Extended Count Key Data: a recording format of data stored on DASDs.
Elevator	Synonym for “I/O Scheduler” in the Linux Kernel.
FCP	Fibre Channel Protocol: transport for the SCSI command set over Fibre Channel networks.
FIFO	First in, First out.
HCTX	Hardware Context of a request queue.
ioctl	input/output control: system call that allows to query device-specific information, or execute device-specific operations.

Glossary ii

IPI	Inter-Processor Interrupt: interrupt an other processor to communicate some required action.
iSCSI	Internet SCSI: transport for the SCSI command set over TCP/IP.
LVM2	Logical Volume Manager: flexible methodes of allocating (non-linear) space on mass-storage devices.
md	Multiple devices support: support multiple physical block devices through a single logical device; required for RAID and logical volume management.
MQ	Short for: Multi-Queue.
Multipathing	Accessing one storage target via multiple independent paths with the purposes of redundancy and load-balancing.
NVMe	Non-Volatile Memory Express: interface for accessing persistent storage device over PCI Express.
RAID	Redundant Array of Inexpensive Disks: combines multiple physical disks into a logical one with the purposes of data redundancy and load-balancing.
RAM	Random-Access Memory: a form of information storage, random-accessible, and normally volatile.
SAS	Serial Attached SCSI: transport for the SCSI command set over a serial point-to-point bus.

Glossary iii

SCSI	Small Computer System Interface: set of standards for commands, protocols, and physical interfaces to connect computers with peripheral devices.
Serial ATA	Serial AT Attachment: serial bus that connects host bus adapters with mass storage devices.
SMR	Shingled Magnetic Recording: a magnetic storage data recording technology used to provide increased areal density.
VFS	Virtual File System: abstraction layer in Linux that provides a common interface to file systems and devices for software.

References i

[1] J. Axboe.
Explicit block device plugging, Apr. 2011.
https://lwn.net/Articles/438256/.

[2] J. Axboe.
Efficient io with io_uring.
https://kernel.dk/io_uring.pdf, Oct. 2019.

[3] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet.
Linux block io: Introducing multi-queue ssd access on multi-core systems.
In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR ’13, pages 22:1–22:10, New York, NY, USA, 2013. ACM.

[4] N. Brown.
A block layer introduction part 1: the bio layer, Oct. 2017.
https://lwn.net/Articles/736534/.

[5] N. Brown.
Block layer introduction part 2: the request layer, Nov. 2017.
https://lwn.net/Articles/738449/.

[6] J. Corbet.
The bfq i/o scheduler, June 2014.
https://lwn.net/Articles/601799/.

References ii

[7] J. Corbet.
Block-layer i/o polling, Nov. 2015.
https://lwn.net/Articles/663879/.

[8] J. Corbet.
The return of the bfq i/o scheduler, Feb. 2016.
https://lwn.net/Articles/674308/.

[9] J. Corbet.
A way forward for bfq, Dec. 2016.
https://lwn.net/Articles/709202/.

[10] J. Corbet.
Two new block i/o schedulers for 4.12, Apr. 2017.
https://lwn.net/Articles/720675/.

[11] J. Corbet.
I/o scheduling for single-queue devices.
Oct. 2018.
https://lwn.net/Articles/767987/.

[12] J. Corbet.
Ringing in a new asynchronous i/o api, Jan. 2019.
https://lwn.net/Articles/776703/.

References iii

[13] K. Desai.
io_uring iopoll in scsi layer, Feb. 2021.
https://lore.kernel.org/linux-scsi/20210215074048.19424-1-kashyap.desai@broadcom.com/T/#.

[14] W. Fischer and G. Schönberger.
Linux storage stack diagramm, Mar. 2017.
https://www.thomas-krenn.com/de/wiki/Linux_Storage_Stack_Diagramm.

[15] E. Goggin, A. Kergon, C. Varoqui, and D. Olien.
Linux multipathing.
In *Proceedings of the Linux Symposium*, volume 1, pages 155–176, July 2005.
https://www.kernel.org/doc/ols/2005/ols2005v1-pages-155-176.pdf.

[16] kernel development community.
Block documentation.
https://www.kernel.org/doc/html/latest/block/index.html.

[17] M. Lei, H. Reinecke, J. Garry, and K. Desai.
blk-mq/scsi: Provide hostwide shared tags for scsi hbas, Aug. 2020.
https://lore.kernel.org/linux-scsi/1597850436-116171-1-git-send-email-john.garry@huawei.com/T/#.

[18] R. Love.
***Linux Kernel Development*.**
Addison-Wesley Professional, 3 edition, June 2010.

References iv

[19] O. Purdila, R. Chitu, and R. Chitu.
Linux kernel labs: Block device drivers, May 2019.
https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html.

[20] O. Sandoval.
blk-mq: abstract tag allocation out into sbitmap library, Sept. 2016.
https://lore.kernel.org/linux-block/cover.1474100040.git.osandov@fb.com/T/#.

[21] K. Ueda, J. Nomura, and M. Christie.
Request-based device-mapper multipath and dynamic load balancing.
In *Proceedings of the Linux Symposium*, volume 2, pages 235–244, June 2007.
https://www.kernel.org/doc/ols/2007/ols2007v2-pages-235-244.pdf.

[22] Western Digital Corporation.
dm-zoned.
https://www.zonedstorage.io/linux/dm/#dm-zoned.

[23] Western Digital Corporation.
Zoned block device user interface.
https://www.zonedstorage.io/linux/zbd-api/.

[24] Western Digital Corporation.
Zoned storage overview.
https://www.zonedstorage.io/introduction/zoned-storage/.

References v

[25] Western Digital Corporation.
zonefs.
https://www.zonedstorage.io/linux/fs/#zonefs.

[26] J. Xu.
dm: support polling, Mar. 2021.
https://lore.kernel.org/linux-block/20210303115740.127001-1-jefflexu@linux.alibaba.com/T/#.