Block Devices, Filesystems And Block Layer Alignment

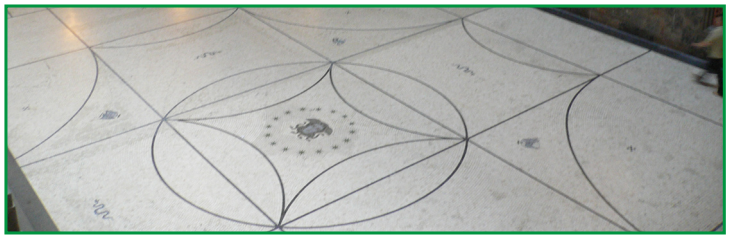
Christoph Anton Mitterer
christoph.anton.mitterer@lmu.de

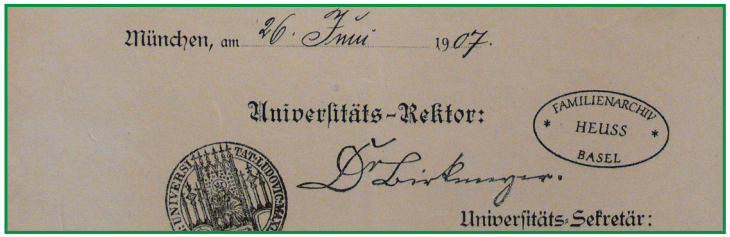
CC BY-SA

Overview

This lecture covers the following chapters:

I. Blocks, Block Devices And Filesystems
   Gives an introduction to blocks, block devices and filesystems and describes common types of them.
II. Block Layer Alignment
   Covers the concepts of block layer alignment, reasons for misalignment and information on how to prevent them for some common systems as well as an overview on the Linux kernel's device topology information.





I. Blocks, Block Devices And Filesystems

Introduction To Blocks

In computing, organising data in blocks is a general and basic technique.
Examples range from most forms of multimedia encodings (for example JPEG, MP3 or H.264) to cryptographic ciphers and even some databases organise their very low level structures in a kind of blocks.

Most storage media and memory (here, the word “page” is typically used) are organised in terms of blocks, although modern concepts like “extents” or “transparent huge pages” makes things a bit more complex on a higher level.

So apart from some exceptions where data is streamed (basically all forms of tape), all the other common types of storage, like hard disk drives, solid state drives and “flash drives or cards” as well as optical discs, are block-addressed.

This lecture focuses on the storage area.

Introduction To Blocks

Blocks have several basic properties:

- The blocks of a given "device" have usually the same size.
  A "basic" and for many areas the smallest block size is 512 B. This used to be the common block size for hard disks but recently drives with 4 KiB showed up, though some of them still behave externally as if they would use 512 B blocks.
- The blocks are directly addressable, that is randomly accessible.
  The contents of a block may be directly accessible or not. For block-organised storage media, the former is usually the case.
  Usually, there is also some latency in accessing a block (for example the "seek time" of hard disks.
- Depending on the "device", data may be only read and/or written as full blocks.
- Depending on the "device", blocks are writeable many times, or just once (for example WORM or non-erasable optical discs).

Filesystems are not block devices themselves but "upon" the laters.
Therefore it is reasonable to view them like another layer.

Introduction To Blocks

Blocks (arranged in a “device”) can be visualised as follows:

Block Devices And Block Layers

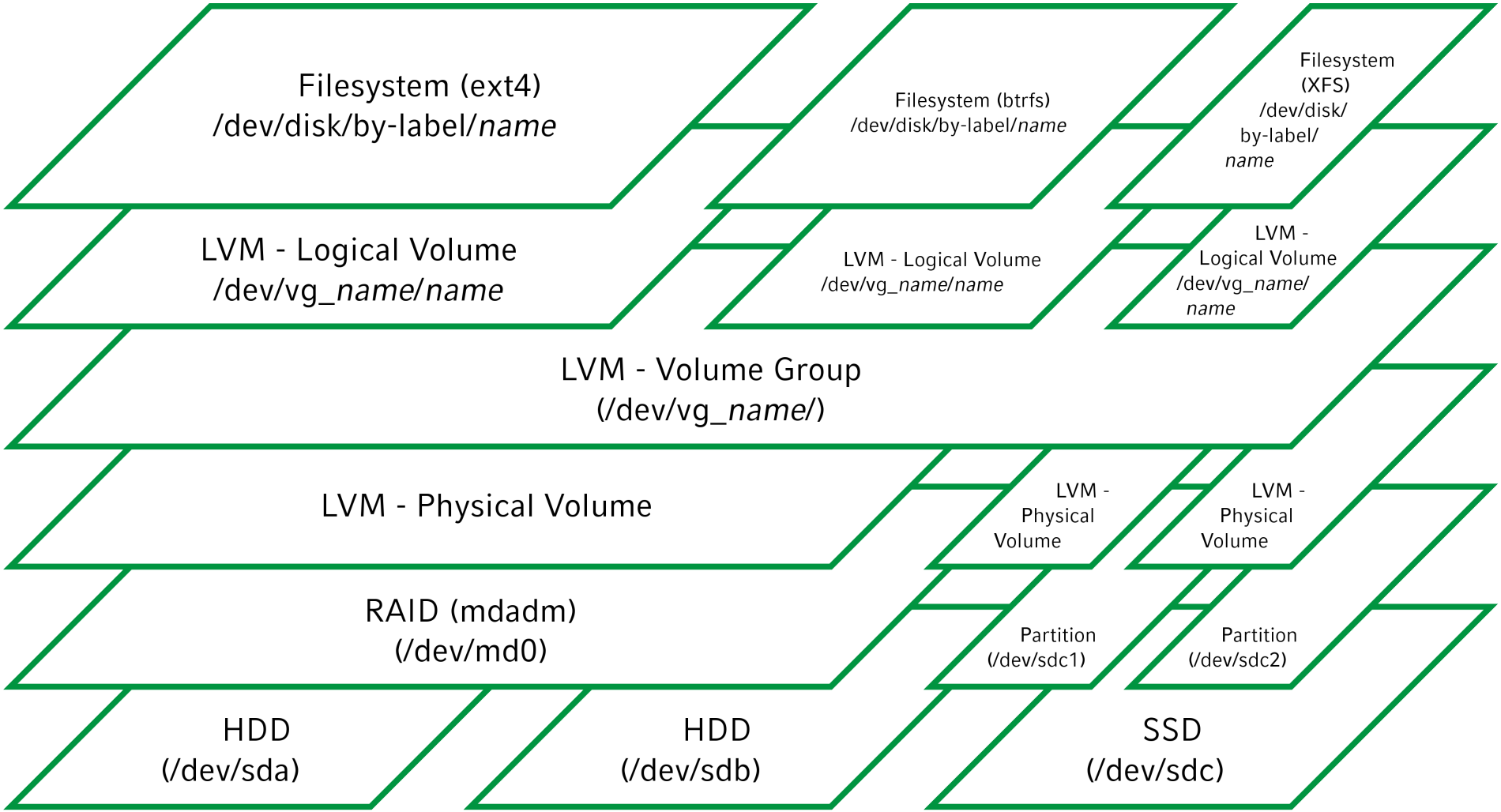
The devices, both the physical and the logical as exported by the operating system kernel, that are organised and addressed in terms of blocks are called “block devices”.
Devices where data is streamed are called “character devices”.

Often, block devices can be stacked, which means that the upper level uses and stores its own data on the lower one.
This works for some physical block devices (for example disk drives that are assembled to one RAID by a hardware controller) and typically for most logical block devices created and handled by the operating system.

Each level in such a stack is called a “block device layer”, or short “block layer”.

Every type of block device implements a special functionality, which is controlled via kernel interfaces and/or the respective hardware controller BIOS.

Block Devices And Block Layers



Common Block Device Techniques

There are a number of general techniques used amongst block devices, including:

- Mapping
  Most types of block devices add meta-data, that shall not be (directly) seen by the upper layer and some types of block devices even distribute the actual data non-sequentially.
  In order that an upper layer "sees" sequentially addressed blocks a virtual addressing is introduced by means of mapping.
  Obviously the mapping costs some performance but this is typically very small and thus neglectable.
- Read Caching And Read Ahead
  Many types of block devices cache data read in either memory or faster storage so that it can be faster retrieved if demanded again.
  Closely related is the technique of "reading ahead", which means that more data than actually requested is automatically read and put into the read cache. More "advanced" algorithms try to predict how much data will be read next and adaptively read ahead.
  Whether read ahead improves performance depends largely on the typical usage patterns so there is no general rule. Obviously, the number of bytes read ahead has a large impact here.

Common Block Device Techniques

- Write Caching
  Typically it is more performant (and may have even other advantages) not to actually write data immediately (to the lower block device, for example the physical media) but to schedule writes in larger chunks.
  A lot of different smart algorithms exist for write caching, usually specifically for the type of block device.
  In order to implement them, another type of cache (again either in memory or on some kind of "faster" storage) is obviously required.
  When the write cache is in volatile memory, the failures (like loss of power) are of course very critical and lead usually to data corruption unless higher levels have added logical means of protection or physical means of protection (for example battery packs) are in place.
  The write algorithms are divided in two policy classes:
  - Synchronous Write ("Write-Through")
    Data is immediately flushed to the next lower layer.
  - Asynchronous Write ("Write-Back" or "Write-Behind")
    Data may be retained in a cache and flushed to disk later, when the algorithm decides this is suitable.

Common Types of Block Devices – Physical Devices

Hard Disk Drives (HDD):

- Block Sizes: typically 512 B, 4 KiB (but many such HDD behave logically as 512 B devices)
- Medium Sizes: ≤ 4 TiB (depends on the technique; smaller for enterprise devices)
- Interfaces: SATA, SAS, Fibre Channel, *legacy*: PATA, SCSI
- Varying seek times depending on how data is distributed and the position of the heads.
- Moving parts leading to mechanical wear.

Solid State Drives (SSD):

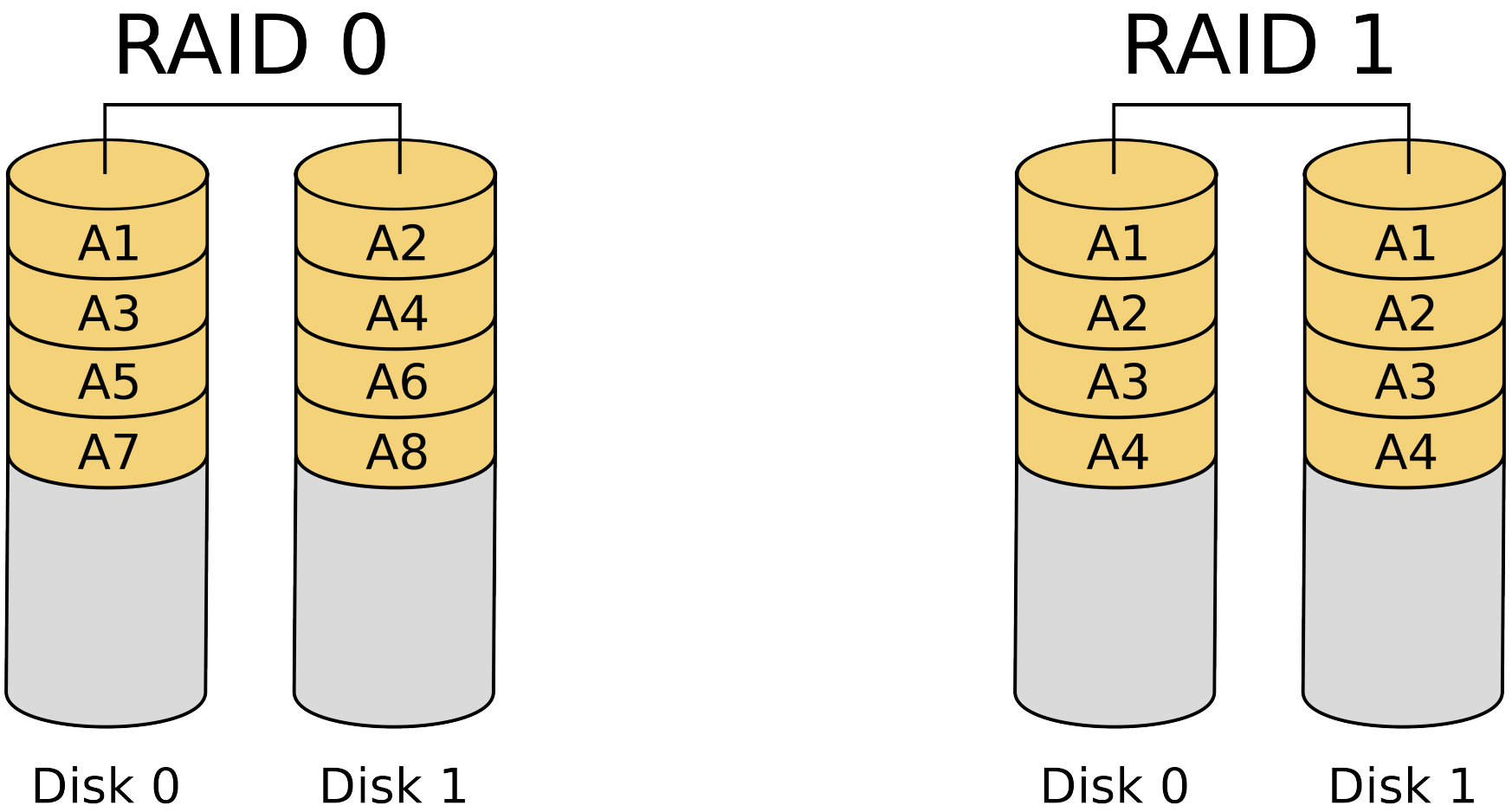
- Block Sizes: typically 512 B, 4 KiB (but many such HDD behave logically as 512 B devices)
- Medium Sizes: ≤ 12 TiB (depends on the technique; smaller for enterprise devices)
- Interfaces: SATA, SAS, Fibre Channel, PCI Express, *legacy*: PATA, SCSI
- Many techniques: typically NAND SLC or MLC, ECC, DRAM-buffered
- Basically much faster than HDD in any respect, but also still more expensive.
- No moving parts, but cells are subject to electrical wear and can only be written a given number of times. Sophisticated wear levelling algorithms are used.
- Cells must be erased before re-written. Therefore always “full” cells are written.

Common Types of Block Devices – RAID

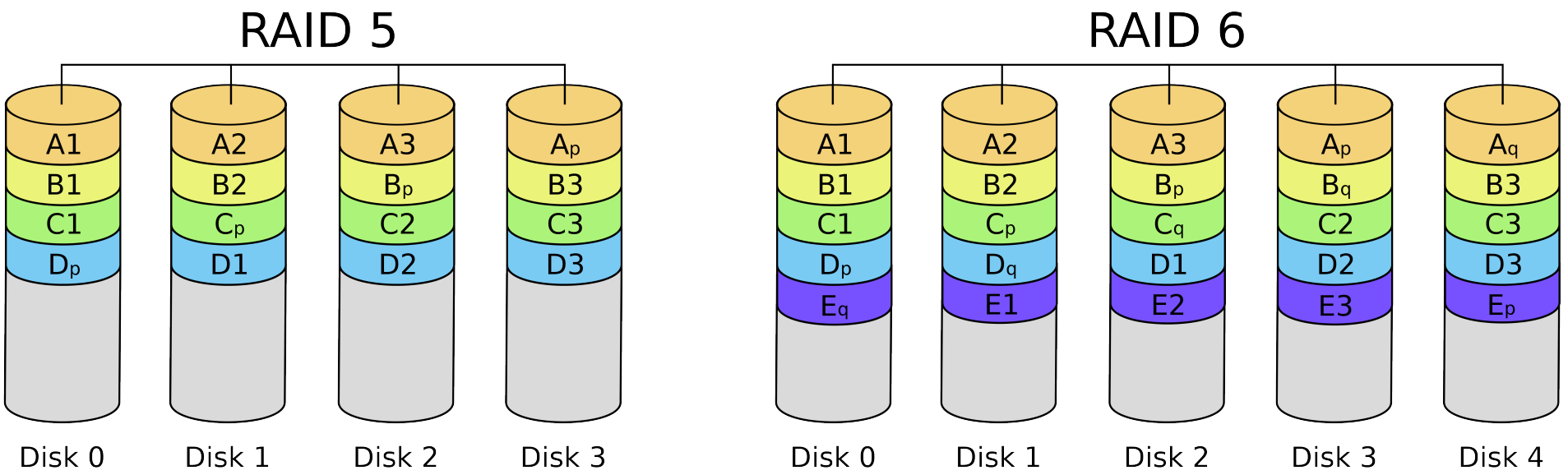
Redundant Array Of Independent Disks:

- Logical combination of other storage media (typically HDD or SDD) for redundancy/resilience, performance or both.
- RAID-Types: hardware, firmware/driver-based ("fake"), software
  (RAID similar features are also found in some modern filesystems or other block device types)
- RAID-Levels: linear, 0, 1, 5, 6, hybrids (for example 10, 50 or 60), *obsolete*: 2, 3, 4
  also "New RAID Classification" by the RAID Advisory Board and non-standard levels
- Typical Techniques: Read Ahead, Adaptive Read Ahead, Write-Through/Write-Back, Hot-Plugging, Hot-Spares, Battery Packs, Scrubbing and Verifying
- Striping: Except in the linear mode, the storage media assembled to a RAID are not "filled" on after each other but "concurrently". Data written is divided into *chunks* of a fixed size, where each chunk is written to the "next" data (not parity) medium.
- Typical chunk sizes are 64 KiB, 128 KiB, 256 KiB, 512 KiB, 1 MiB
- It depends on the respective RAID-implementation and also on the RAID-level, but usually one must expect that always "full" chunks are read and written.
  Therefore, the chunk size may greatly influence the performance of a RAID, depending on the respective use case.
- The stripe size is usually the size of one stripe with its data and parity chunks.

Common Types of Block Devices – RAID



Common Types of Block Devices – RAID

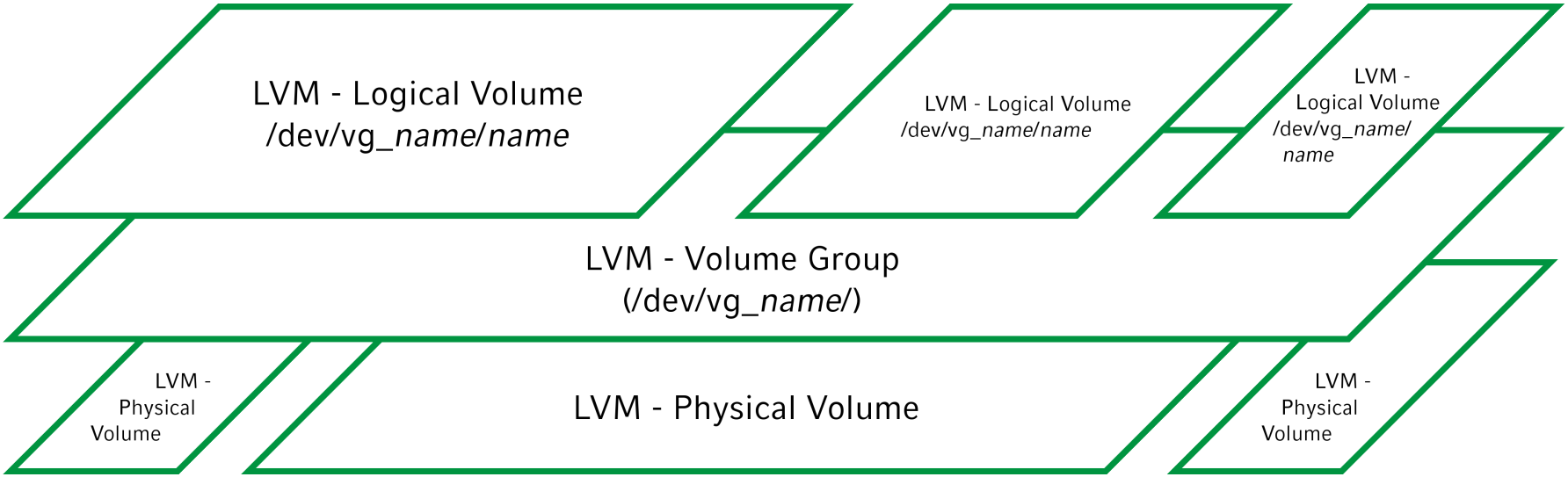


Common Types of Block Devices – LVM

Logical Volume Manager:

- A "front-end" to Linux' device-mapper, that allows management of arbitrary block devices as volumes.
- Physical Volumes (PV): These are the "underlying" block devices used by LVM for storing the data.
- Volume Groups (PV): PV are organised in VG, which have a number of properties including a chunk size and an allocation policy (that is how chunks from the PV are distributed to underlying LV).
  Each VG can have multiple PV, but each PV must belong to exactly one VG.
- Logical Volumes (LV): The block devices exported to be used by "upper" layers.
- LVM allows to combine or divide block devices to other block devices, which gives it features known from the RAID levels linear and 0 and from partitioning.
  PV and LV can be added/removed to/from existing VG.
  LVM also implements advanced features like clustering, snapshots, striping or mirroring.
- Data is organised in extents (default size 4 MiB), which are however not fully read and written, as this is usually the case with RAID chunks.

Common Types of Block Devices – LVM



Common Types of Block Devices – Miscellaneous

Partitions:

- Logical separation of a block device in several other block devices.
- Different types of partition labels (or partition tables) including: DOS, BSD Disklabel, GUID Partition Table
- Depending on the type of partition label, there are several limitations, for example the DOS type cannot handle partitions $\gtrsim$ 2 TiB, the number of partitions is limited and they cannot be moved.
- In most cases not needed anymore, as LVM is much more flexible in any way.

dm-crypt:

- A “front-end” to the device-mapper providing on-disk-encryption.
- Strong algorithms and cipher modes tailored towards on-disk-encryption (for example XTS).

dm-multipath:

- Several paths (“connections”) to the same lower level block device for redundancy.

Loop devices:

- Maps a file to a block device.

Filesystems

Filesystems lay on top of block devices and export a file hierarchy to the user space, in which data is organised as files and not longer just "meaningless" blocks. Thereby, filesystems hide the block layout and organisational details from the user space.

Some properties of filesystems:

- A lot of different kinds of global and per-file meta-data, including the "normal" POSIX properties as well as XATTR and ACL.
- Files are internally organised as blocks or – on some newer filesystems – alternatively as extents (larger contiguous and differently sized areas of blocks, reserved for parts of a given file).
- Sophisticated algorithms for (amongst others) IO-caching and delayed writes, blocks/extents allocator algorithms, et cetera.

Filesystems

Some types of filesystems:

- "Normal" Filesystems
  btrfs, ext2/3/4, XFS, JFS, ReiserFS, Reiser4, ZFS, UFS
- Media-Centric Filesystems
  UDF, ISO9660, JFFS2, LogFS
- Pseudo Filesystems
  procfs, sysfs, swap
- Special Filesystems
  tmpfs, aufs, romfs, SquashFS
- Network- And Cluster Filesystems
  NFS, CIFS, SMB, GFS2, GPFS, OCFS2, AFS, GlusterFS, Lustre, GFS, XtreemFS, Ceph

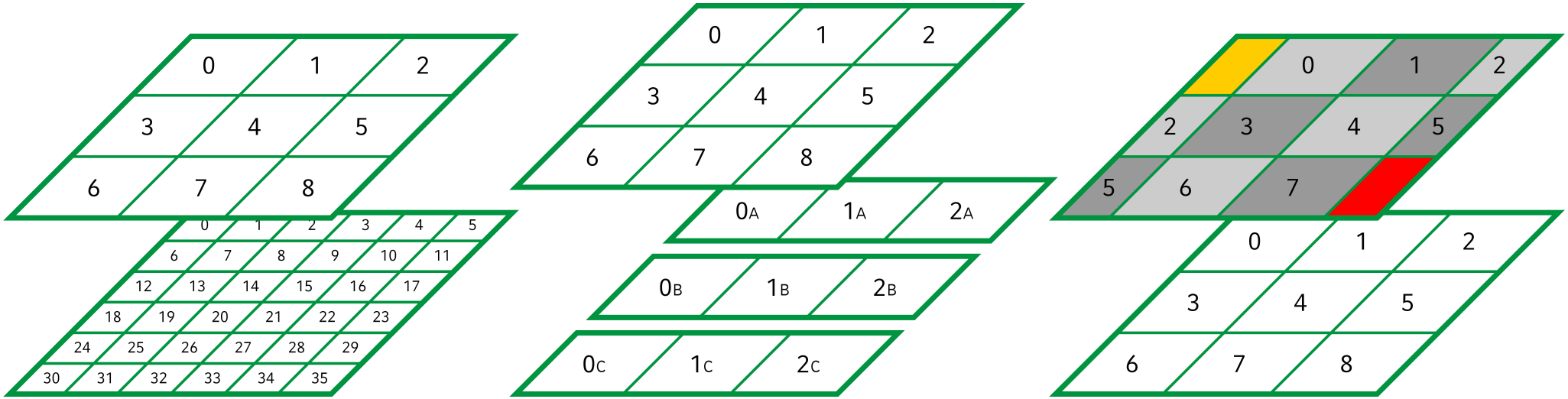
Filesystems may be implemented in user space via FUSE, for example:
davfs2, SSHFS, GlusterFS, GmailFS, et cetera

II. Block Layer Alignment

Introduction To Block Layer Alignment

The following general properties are inherent to block devices:

- They have a total size.
- They organise their data in structures like blocks, chunks, stripes and extents, where the respective structures of different devices (and therefore on different block layers) may have different sizes.
- They may change the addressing of blocks via mapping, thereby arranging them differently (for example striped or randomly instead of contiguously).
- They may add meta-data in form of headers, footers or within their block space.

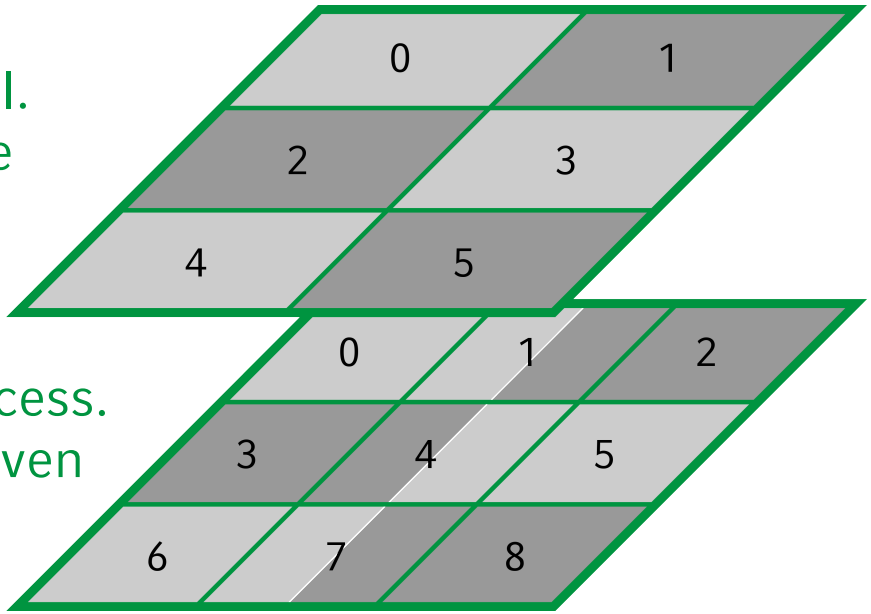


Misalignment – Structure Sizes Are Not Multiples

Generally, the size of a block layer's structures (for example blocks or chunks) should be a (integral) multiple of the size of the respective lower block level's structures. If not, alignment problems may occur.

Scenario: Blocks have 1,5 × the size of lower level's blocks.

- As noted, blocks may be "fully" read/written. Therefore, when a block is accessed on the upper level, more than actually necessary are accessed on the lower level.
- Example: Block 0 is accessed on the upper level. Then blocks 0 and 1 need to be accessed on the lower level. The 2nd half of block 1 was not required.
- Throughput-wise not that big problem on streaming (if caching works) but on random-access. Moreover, the lower block 1 may be accessed even twice, when the upper block 1 is read, too. In any case, unnecessary IOPS may be produced.

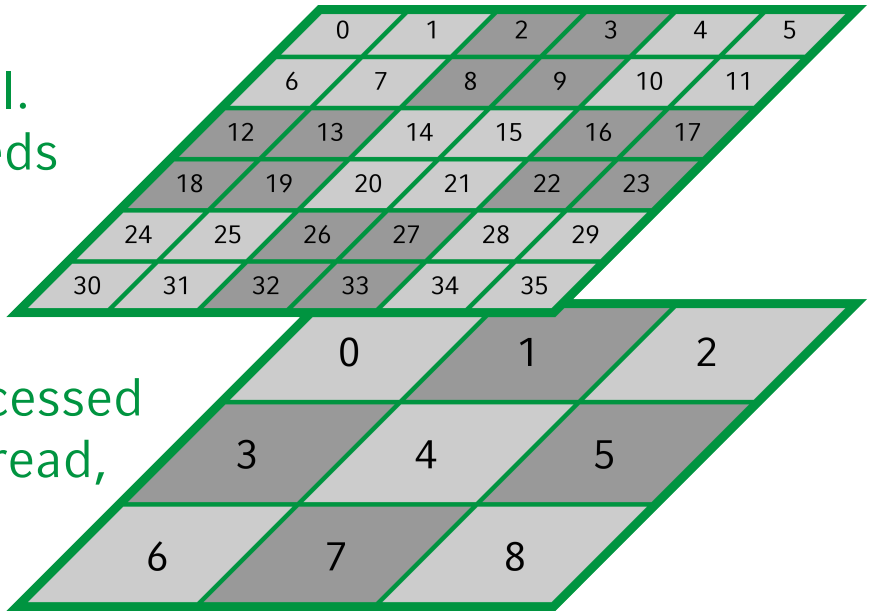


Misalignment – Structure Sizes Are Not Multiples

Generally, the size of a block layer's structures (for example blocks or chunks) should be a (integral) multiple of the size of the respective lower block level's structures.
If not, alignment problems may occur.

Scenario: Blocks have ¼ × the size of lower level's blocks.

- As noted, blocks may be "fully" read/written. Therefore, when a block is accessed on the upper level, more than actually necessary are accessed on the lower level.
- Example: Block 0 is accessed on the upper level. Then block 0, of which ¾ are not required, needs to be accessed on the lower level.
- Throughput-wise not that big problem on streaming (if caching works) but on random-access. Moreover, the lower block 1 may be accessed even twice, when the upper block 1, 6 or 7 are read, too. In any case, unnecessary IOPS may be produced.



Misalignment – Structure Sizes Are Not Multiples

Some hints:

- Most tools prevent creating structure sizes that are not powers of 2, or warn at least. But they usually do not warn if you use sizes on a higher level, that are smaller than those of lower levels.
- The filesystem's block size is per default (ext2/3/4 uses for example 4 KiB) often much smaller than the chunk size (typically starts at 64 KiB) of an underlying RAID.
  It may generally be reasonable to increase the filesystem's block size when mainly big files are used.
- Whether blocks are "fully" read/written depends on the type of block device and often on the specific model or implementation.
  - HDD and SSD and filesystems typically access "full" blocks.
  - For RAID this is highly dependent on the model/implementation.
    In principle a RAID should not need to read "full" chunks under normal operation. But in general: check the respective documentation!
  - LVM does not access full extents under normal operation (with the exceptions when using snapshots and copy-on-writes happen).

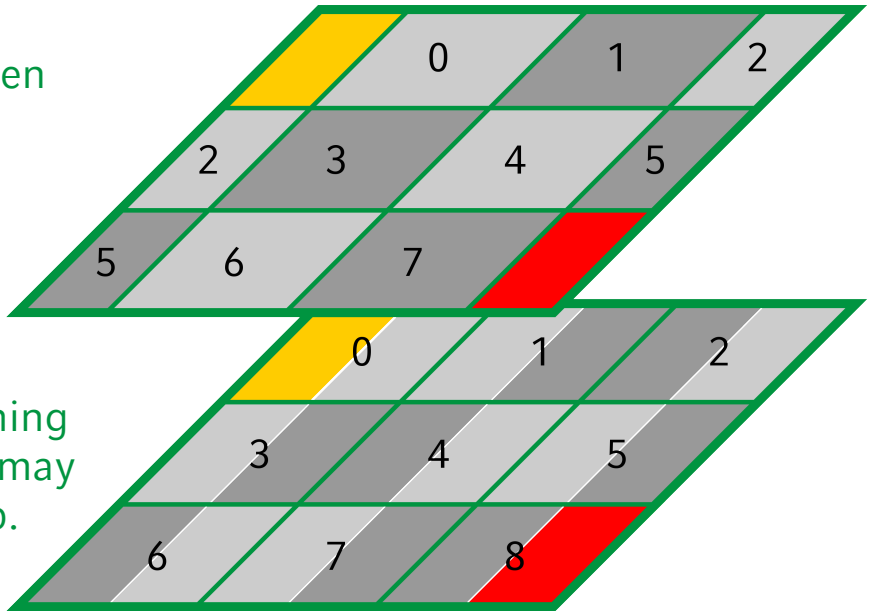
Misalignment – Header Shifts Actual Data

Many types of block devices add meta-data in form of headers, which are not "seen" by the addressing "exported" to higher levels.
To avoid misalignments created by the shift through the headers, padding must be generally used.

Scenario: A block device has a header but does not align the actual data via padding.

Analogous to the previous misalignment cases:

- As noted, blocks may be "fully" read/written. Therefore, when a block is accessed on the upper level, more than actually necessary are accessed on the lower level.
- Example: Block 0 is accessed on the upper level. Then blocks 0 and 1 need to be accessed on the lower level. The 1st half of block 0 and the 2nd half of block 1 were not required.
- Throughput-wise not that big problem on streaming (if caching works) but on random-access. Moreover, the lower block 1 may be accessed even twice, when the upper block 1 is read, too. In any case, unnecessary IOPS may be produced.

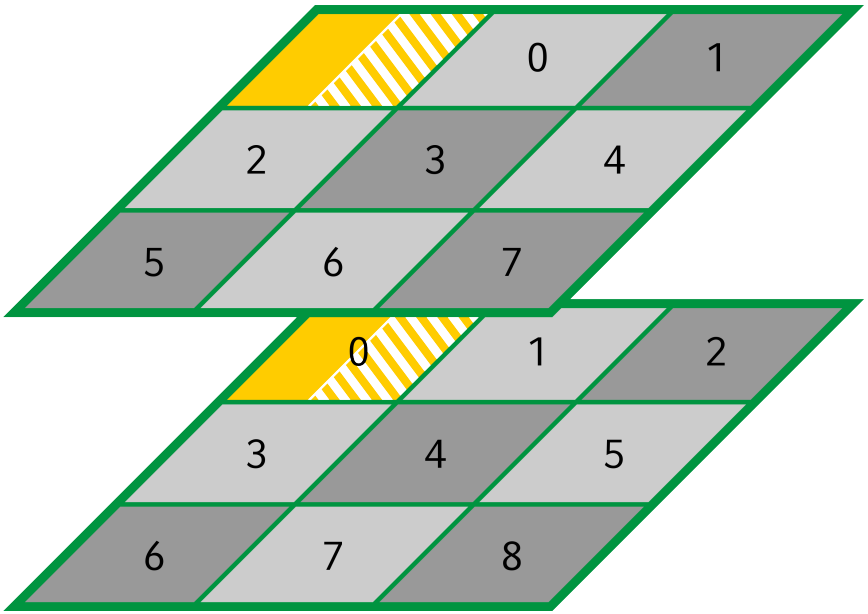


Misalignment – Header Shifts Actual Data

Many types of block devices add meta-data in form of headers, which are not "seen" by the addressing "exported" to higher levels.
To avoid misalignments created by the shift through the headers, padding must be generally used.

Scenario: A block device has a header and correctly aligns the actual data via padding.

None of the previously described problems may occur.



Misalignment – Header Shifts Actual Data

Some hints:

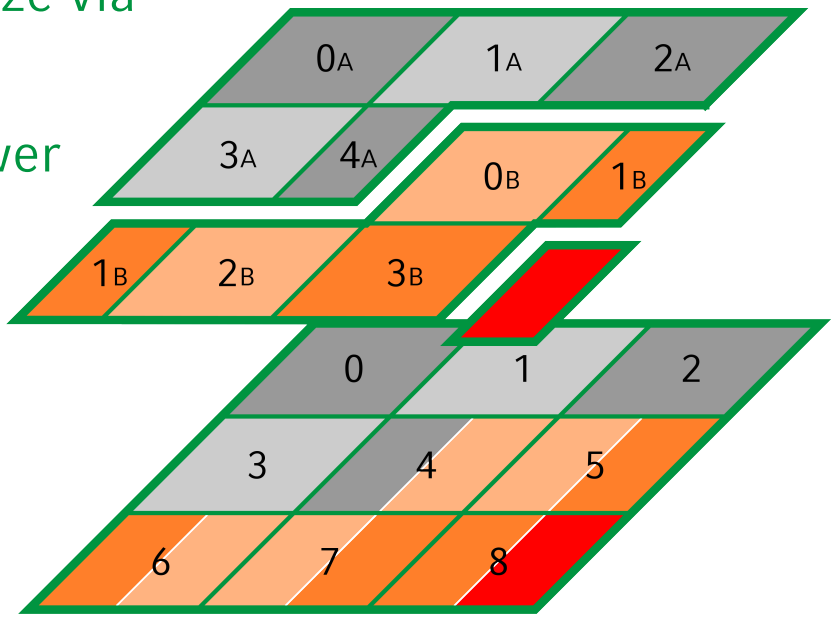
- Hardware RAID have a lot of meta-data but need usually not be aligned. The "global" meta-data is stored in the controller itself and the parity data is of the same size as the actual data chunks and therefore "automatically" aligned if these are.
- The mdadm software RAID from Linux may be used with four different super-block formats:
  - 0.9 and 1.0
    Stored at/near the end of the underlying block devices. Alignment is not necessary.
  - 1.1 and 1.2
    Stored at/near the beginning of the underlying block devices. Alignment is necessary.
- Partitions and LVM need to be aligned.

Misalignment – Block Device Total Size Is Not A Multiple

Generally it is possible, that multiple block devices "lay" upon one block device.
Even if the single structures of the respectively "preceding" block device (on the same layer) are correctly aligned, misalignments may be created for a block device when the total size of the respectively "preceding" block device is not a (integral) multiple of the lower level block device's structures.
Generally, the total size of a block device should be a (integral) multiple of the lower level block device's structures, or filled to such a size via padding.

Scenario: The 1st block device is aligned to the lower layer, but its total size is not a (integral) multiple of the lower layer's structures and padding is not used.

Effects and problems analogous to the previous misalignment cases.

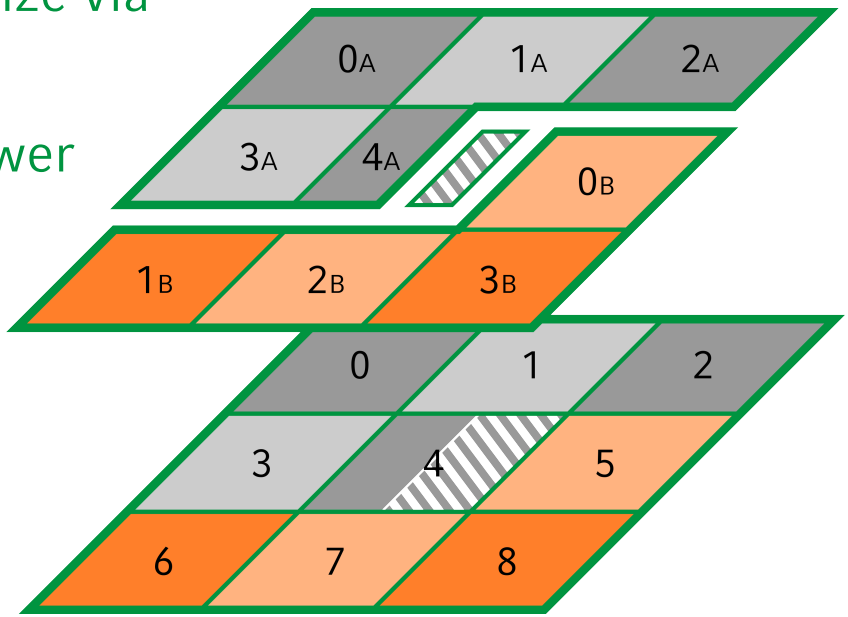


Misalignment – Block Device Total Size Is Not A Multiple

Generally it is possible, that multiple block devices "lay" upon one block device.
Even if the single structures of the respectively "preceding" block device (on the same layer) are correctly aligned, misalignments may be created for the a block device when the total size of the respectively "preceding" block device is not a (integral) multiple of the lower level block device's structures.
Generally, the total size of a block device should be a (integral) multiple of the lower level block device's structures, or filled to such a size via padding.

Scenario: The 1st block device is aligned to the lower layer and its total size is not a (integral) multiple of the lower layer's structures, but padding is used.

None of the previously described problems may occur.

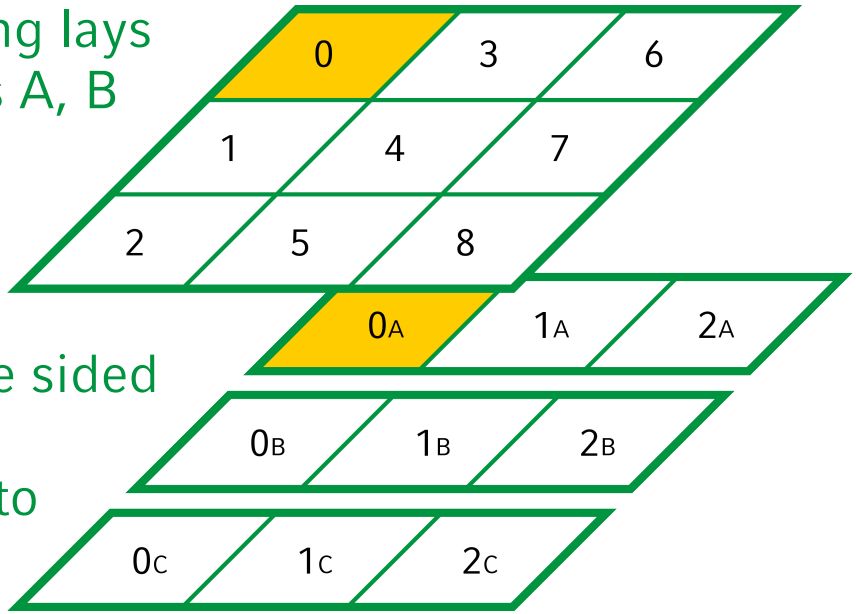


"Misalignment" – Unbalanced Global Meta-Data Spreading

Many types of block devices and every filesystem contain (global) meta-data, which are usually placed near their beginning or end (as header or footer).
If multiple block devices "lay" below, it can easily happen that parts (or even all) of that meta-data end up on only some (or even exactly one) of the lower block devices, which is typically bad for performance and in case of physical devices the wear.
Some block devices and filesystems offer options for spreading (global) meta-data.

Scenario: A filesystem without meta-data spreading lays upon a RAID 0, composed of three physical drives A, B and C.

- "By chance", the global meta-data is fully on drive A.
- Any read or write of the meta-data will put a one sided load on drive A.
- Both, read and write caching mitigate this only to some extent.

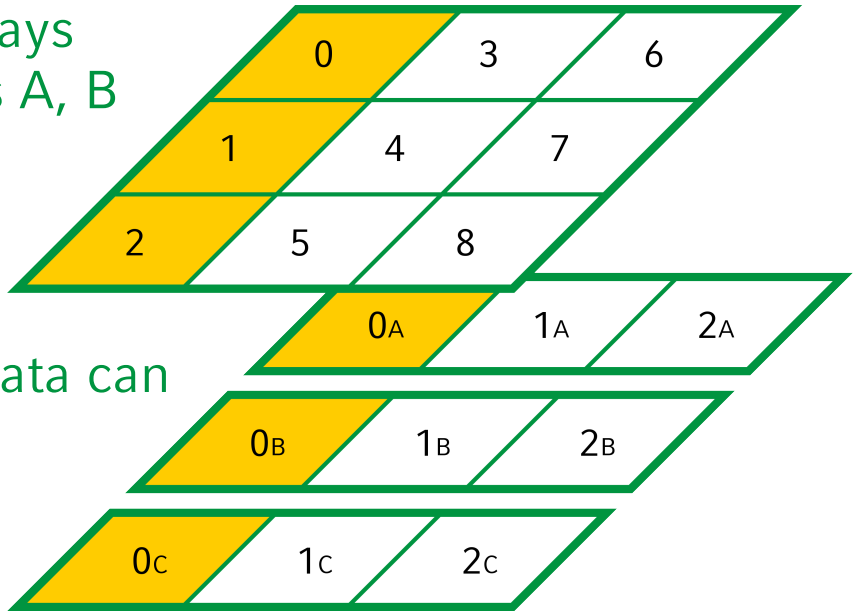


"Misalignment" – Unbalanced Global Meta-Data Spreading

Many types of block devices and every filesystem contain (global) meta-data, which are usually placed near their beginning or end (as header or footer).
If multiple block devices "lay" below, it can easily happen that parts (or even all) of that meta-data end up on only some (or even exactly one) of the lower block devices, which is typically bad for performance and in case of physical devices the wear.
Some block devices and filesystems offer options for spreading (global) meta-data.

Scenario: A filesystem with meta-data spreading lays upon a RAID 0, composed of three physical drives A, B and C.

- The global meta-data is spread over drives A, B and C.
- Reads and sometimes even writes of the meta-data can be balanced between the drives A, B and C.



"Misalignment" – Unbalanced Global Meta-Data Spreading

Some hints:

- When RAID is used, layers above are generally prone to unbalanced spreading of global meta-data.
- When LVM is used, layers above may be prone to unbalanced spreading of global meta-data.
  This is especially the case, if its extent size or the total sizes of PV or LV is not a multiple of the lower layer's structure sizes.
  Care must also be taken to consider the different allocation policies (the "order" in which chunks from underlying PV are distributed to LV).

How-To Prevent Misalignment – mdadm Software RAID

mdadm software RAID is automatically aligned for most "normal" cases, but difficult to completely align for arcane setups (for example mdadm software RAID on top of LVM).
Exact details on the placement of the actual data start and end as well as details on the size and positioning of the super-block can be found in the `md(8)` manpage.

The following `mdadm` options are of special interest:

- `--metadata`
  The type of super-block format to be used.
- `--chunk`
  The chunk size to be used.
- `--size`
  The space used from each underlying block device and thus indirectly the total space of the RAID.
- Other possibly interesting options include:
  `--layout`

How-To Prevent Misalignment – LVM

The following `pvcreate` options are of special interest:

- `--dataalignmenta`
  Aligns the start of the actual data to this offset (or a multiple, if required).
- `--dataalignmentoffset`
  An additional shift of the data area.

The following `vgcreate` options are of special interest:

- `--physicalextentsize`
  Sets the value of the extent size used by the respective VG.

The following `lvcreate` options are of special interest:

- `--extents`
  The size of the LV in extents. Preferred over `--size`, which sets the size in bytes.
- `--contiguous`
  Whether contiguous extent allocation should be performed or not.
- Other possibly interesting options include:
  `--readahead`, `--type`, `--stripes`, `--stripesize` and `--mirrors`

How-To Prevent Misalignment – LVM

The following general `lvm` options are of special interest:

- `--alloc`
  Sets the extent allocation policy to one of `contiguous`, `cling`, `normal`, `anywhere` or `inherit`.

How-To Prevent Misalignment – Miscellaneous

Partitions:
Can be aligned manually by aligning the partition start and end addresses.

dm-crypt:
The following `cryptsetup` options are of special interest:

- `--align-payload`
  Aligns the start of the actual data to a given multiple of 512 B.
- Other possibly interesting (when LUKS is not used) options include:
  `--size` and `--offset`

Loop device:
Basically, for a loop device to be aligned, the underlying filesystem must be aligned.
If this is not the case, a compensation may be possible with the `--offset` option.

- `--offset`
  Shifts the start of the loop device into the file.
- `--sizelimit`
  Sets the size of the device.

How-To Balance Global Filesystem Meta-Data – ext2/3/4

`mke2fs` and `tune2fs` provide the following options:

- `-E stride=`value
  The RAID's chunk size in number of filesystem blocks.
- `-E stripe_width=`value
  The size of the data parts of the RAID's stripes in filesystem blocks.
  That is the number of data chunks per stripe multiplied with the value from the `-E stride` option.

How-To Balance Global Filesystem Meta-Data – ext2/3/4

Examples:

- RAID 6 with a chunk size of 256 KiB and 10 drives in total (8 data drives, 2 parity drive, 0 hot spares); filesystem with a block size of 4 KiB
  `-E stride=`(256 KiB ÷ 4 KiB = `64`)`,stripe_width=`(64 · 8 = `512`)
- RAID 6 with a chunk size of 256 KiB and 10 drives in total (7 data drives, 2 parity drive, 1 hot spare); filesystem with a block size of 4 KiB
  `-E stride=`(256 KiB ÷ 4 KiB = `64`)`,stripe_width=`(64 · 7 = `448`)
- RAID 60 with a chunk size of 256 KiB and 10 drives in total (6 data drives, 4 parity drive, 0 hot spares); filesystem with a block size of 4 KiB
  `-E stride=`(256 KiB ÷ 4 KiB = `64`)`,stripe_width=`(64 · 6 = `384`)

How-To Balance Global Filesystem Meta-Data – XFS

`mkfs.xfs` provides the following options:

- `-d su=`value
  The RAID's chunk size in bytes.
  `sunit=`value is an alternative form, where the value has to be specified in 512 B blocks.
- `-d sw=`value
  The size of the data parts of the RAID's stripes in bytes.
  `swidth=`value is an alternative form, where the value has to be specified in 512 B blocks.

How-To Balance Global Filesystem Meta-Data – XFS

Examples:

- RAID 6 with a chunk size of 256 KiB and 10 drives in total (8 data drives, 2 parity drive, 0 hot spares); filesystem with a block size of 4 KiB

  ```
  -d su=256 -d sw=8
  ```

- RAID 6 with a chunk size of 256 KiB and 10 drives in total (7 data drives, 2 parity drive, 1 hot spare); filesystem with a block size of 4 KiB

  ```
  -d su=256 -d sw=7
  ```

- RAID 60 with a chunk size of 256 KiB and 10 drives in total (6 data drives, 4 parity drive, 0 hot spares); filesystem with a block size of 4 KiB

  ```
  -d su=256 -d sw=6
  ```

Linux' Device Topology Information

Beginning with recent versions (starting with about 2.6.34) of the Linux kernel functionality was added to determine topology information for block devices, including the alignment offset, the physical and logical block sizes, as well as the minimum and optimal IO-sizes.

This can be used by userland tools to automatically set the respective values.

The device topology information is also exported via sysfs:

- `/sys/block/block-device[/partition]/alignment_offset`
- `/sys/block/block-device/queue/physical_block_size`
- `/sys/block/block-device/queue/logical_block_size`
- `/sys/block/block-device/queue/hw_sector_size`
- `/sys/block/block-device/queue/minimum_io_size`
- `/sys/block/block-device/queue/optimal_io_size`

Documentation can be found in `./Documentation/ABI/testing/sysfs-block` the Linux kernel.

Automatic Alignment

Beginning with recent Linux kernels some recent userland tool versions may be capable of using the kernel's device topology information to automatically detect the correct settings for alignment in some scenarios.

Examples:

- LVM
  Recent versions of `lvm` try to determine any underlying mdadm software RAID, alignment to their chunk sizes and alignment of LVM's actual data start.
  The following `lvm.com` options are of special interest:
  `md_component_detection`, `md_chunk_alignment`, `data_alignment_detection`, and `data_alignment_offset_detection`
- dm-crypt
  Recent versions of `cryptsetup` try to determine alignment of the actual data start.
- Partitions
  Recent versions of GNU Parted try to align partitions, when the `--align=optimal` option is used.
  util-linux' fdisk and GNU fdisk have no support, so far.

Automatic Alignment

General rule: Any automatically determined alignment values should be manually verified!

Literature

- http://people.redhat.com/msnitzer/docs/io-limits.txt
- https://ata.wiki.kernel.org/articles/a/t/a/ATA_4_KiB_sector_issues_d4b8.html
- https://raid.wiki.kernel.org/

Finis coronat opus.