Western Digital®

zonefs: Features Roadmap

Damien Le Moal

Distinguished Engineer, System Software Group, Western Digital Research

Sep 14th, 2022

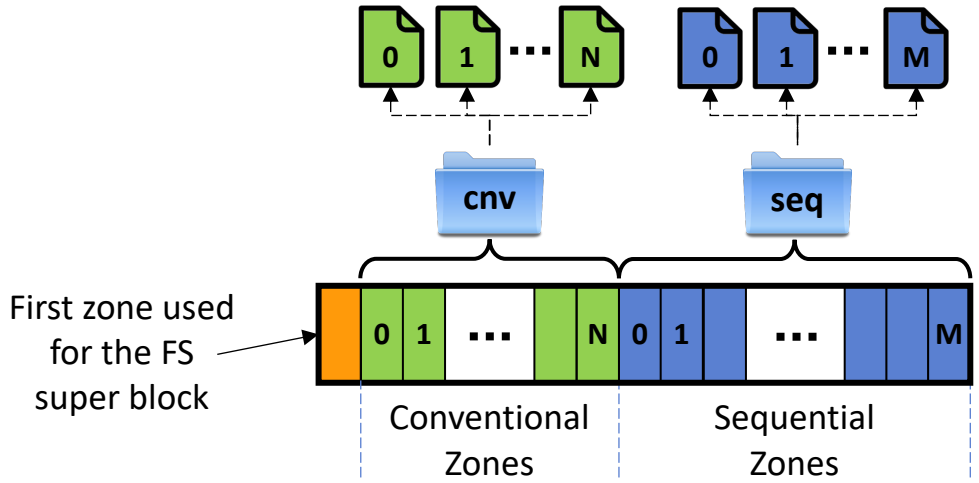
Outline

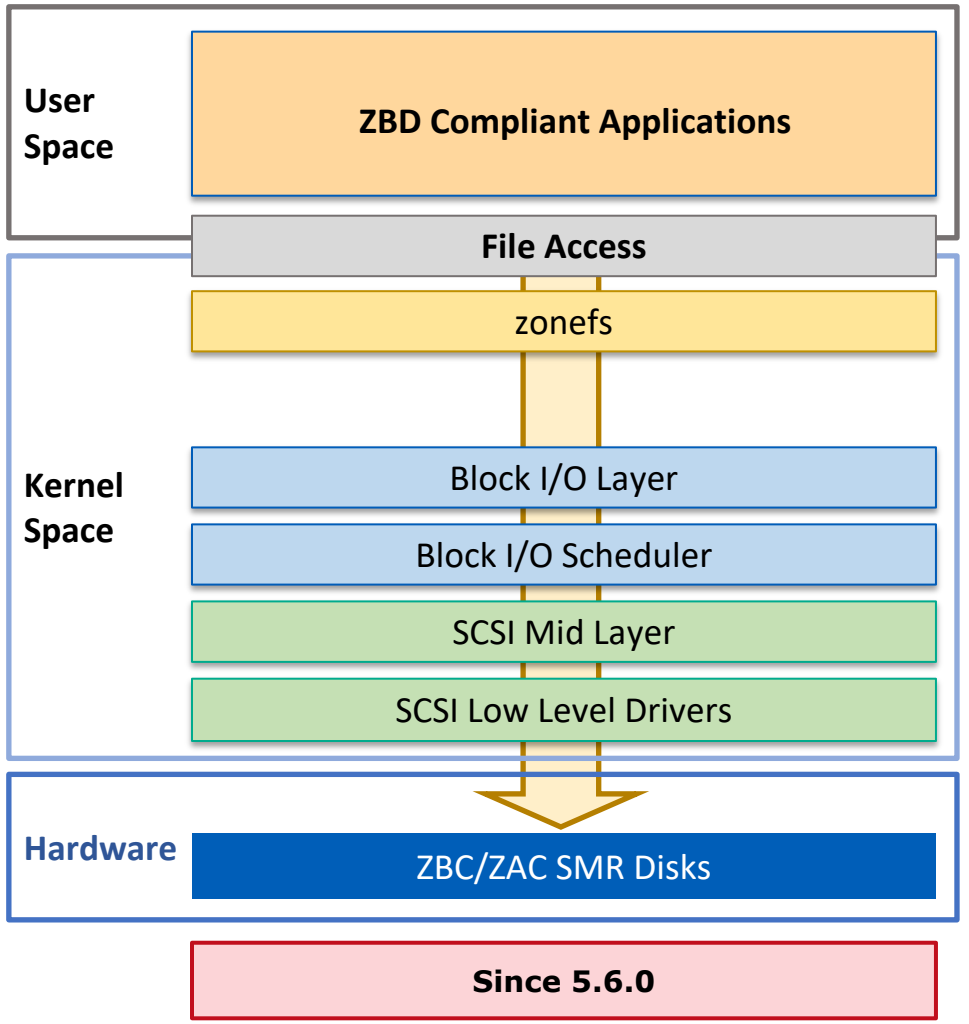
- Zonefs overview and its position in Linux zoned storage ecosystem
- Zonefs recent (ish) fixes and updates
- Zonefs features roadmap
  - Read IO tail latency improvements
  - Reducing memory usage
  - Asynchronous zone append
  - Allowing buffered writes

zonefs Overview

Expose each zone of a zoned device as an append-only file

- zonefs exposes the zones of a zoned device as files
  - Files are grouped per zone type in different sub-directories
  - Files of sequential write required zones cannot be written randomly: O_APPEND writes only (append only file)
- Seamless integration of zone commands within regular file system calls
  - E.g. truncate(), ftruncate() -> zone reset or zone finish

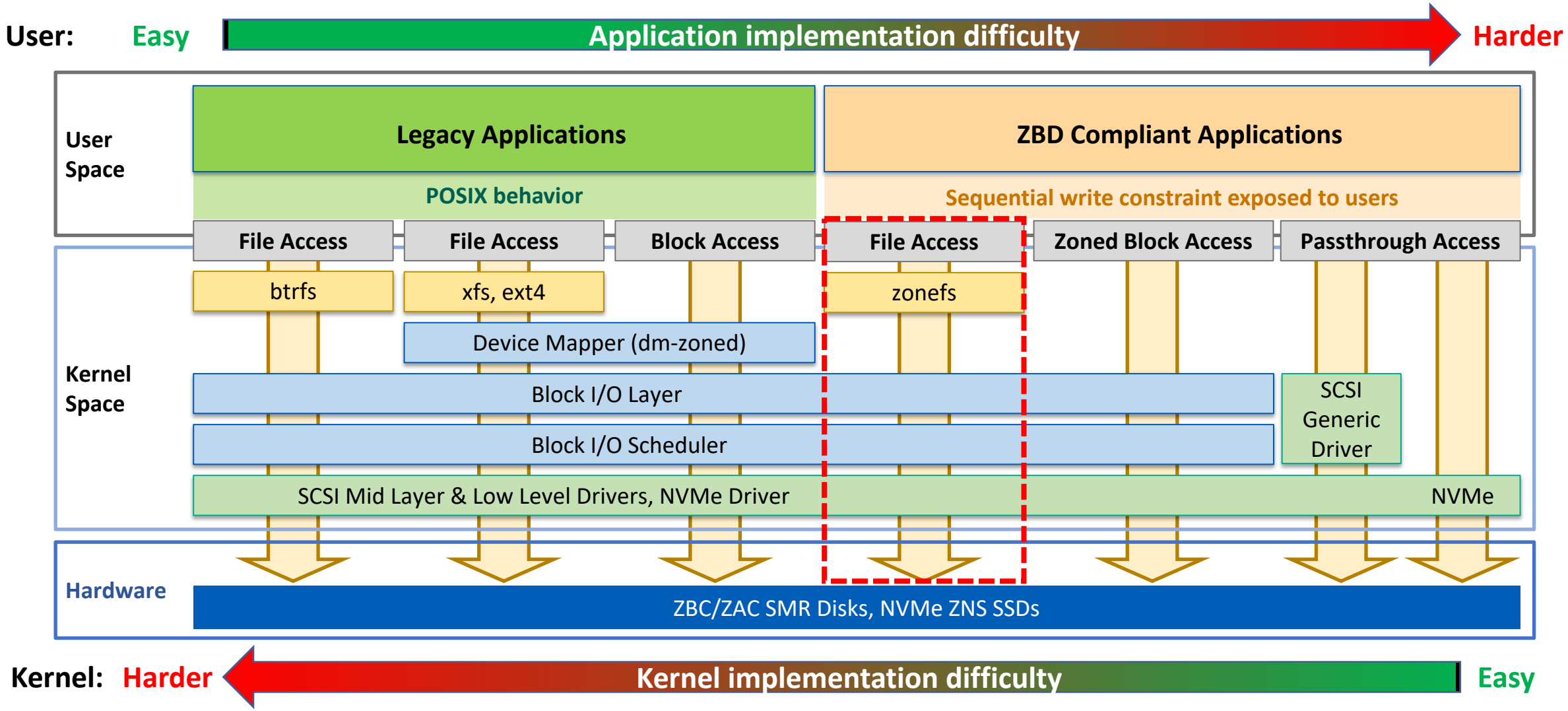




https://zonedstorage.io/linux/fs/#zonefs

zonefs Position In the Zoned Storage Ecosystem

Facilitates implementation of zone compliant applications



Zonefs Recent (ish) Updates

Bug fix and active zone management through explicit_open option

- Nasty read-ahead bug fix
  - Could cause an infinite loop during read-ahead processing
    - Due to an incorrect implementation of the iomap begin method
- Improved handling of the explicit_open mount option
  - Number of files that can be open for writing is limited by both the maximum open zone limit *and* the maximum active zone limit
  - A file open for writing is kept “active” even if it is empty or full, that is, even if its underlying zone is not active
    - Ensure that the user can always re-start writing the file at any time
    - Maintains the guarantee that if a file can be open for writing, then it can be written
      - Assuming a healthy device (of course)
- Sysfs attributes for max open files, max active files, current number of open & active files, etc
  - Regardless of explicit_open mount option use

Planned Improvements and New Features

Performance (IO latency) and ease of use

- IO tail latency improvements
  - Lower tail latency of read operations executed in parallel with zone operations
- Reducing memory usage
  - On-demand inode allocation with open()
- Asynchronous zone append
  - Enable the use of REQ_OP_ZONE_APPEND BIOs for asynchronous write IOs
- Allowing buffered writes
  - Remove the O_DIRECT write constraint

Performance related improvements

Ease-of-use related improvements

Read IO Tail Latency Improvements

Switch to "unusual" locking model to reduce read IO tail latency

- Problem: read IO operations may be delayed if a concurrent zone operation is also being executed, e.g. a ftruncate() call changing a zone file size to its maximum possible value
  - Zone finish operation transitioning the zone file to full state
- "normal" inode locking calls for ftruncate() write locking the inode (because of the file size change) and the read operation read locking
  - While the zone finish operation is ongoing, read operations must wait
- This is unnecessary: zone finish can be switched to a read lock
  - Zone finish operation does not affect the zone data, nor does it change the file block allocation
    - All file blocks are always "pre-allocated" (allocation implied from the LBA range covering the zone capacity)
  - Truncate mutex will keep serializing truncate operations
  - Concurrent write operations may:
    - Either wait for the zone finish holding the read lock and then fail (that is the user's fault)
    - Proceed first and the zone finish execute normally, eventually even being a nop (still a weird pattern that is the user's fault)

Reducing Memory Usage

On-demand inode initialization

- Currently:
  - All file inodes and directory entries are initialized and cached on mount
    - No dynamic allocation of inodes and directory entries
  - Unused zone files consume memory (inode and dentry)
    - 100,000+ zones on latest generation SMR drives
    - Significant memory usage
- Optimization: on-demand inode initialization
  - On open(), use .get_inode()
    - Allocate inode
    - Do report zones to get the file size/wp location
  - Directory entries not really needed: inode “number” can be inferred directly from file name
    - Save more memory but cannot use generic_read_dir() / dcache_readdir()
      - Need special code
- A lot more code needed and open() performance hit... Is it worth it ?

Asynchronous zone append

Allows a user to run zonefs without the mq-deadline scheduler

- Planned semantic
  - File *not* open with O_APPEND: regular write operations (REQ_OP_WRITE)
  - File open with O_APPEND: zone append write operations (REQ_OP_ZONE_APPEND)
    - Written file offset is returned to the user as the AIO result
- What we need:
  - Ability to return a 64-bits offset (written offset)
    - Trivial with legacy AIOs, a little more difficult with io_uring but now possible thanks to the addition of large CQEs
    - Any FS would gain the ability to return the written offset for O_APPEND writes
  - Adding an iomap submit_bio hook to zonefs to issue zone append operations
    - These BIOs cannot be split: one AIO must be exactly one BIO
    - This implies 2 choices:
      - (1) switch back to regular writes if any AIO is too large for a zone append and wait for the completion of any on-going zone append write before issuing the regular writes, or (2) fail the io_submit() call
      - (1) is preferred to maintain backward compatibility but is less predictable for the user, e.g. “can I get rid of mq-deadline ?” becomes hard to determine. Mount options ? Thoughts ?

Allowing Buffered Writes

Remove O_DIRECT write constraint

- Planned semantic
  - Write *must* remain aligned to file blocks (sectors)
    - Last sector update problem: read-modify-write is not possible
  - O_SYNC like writes, always
    - No guarantees from the page cache that delayed dirty page writes are sequential
- Fairly straightforward implementation
  - Write() context needs to: (1) allocates a folio, (2) copies new data into it, (3) submit the folio for writing, (4) add the folio to the page cache on completion
    - All under the inode write lock
    - Handling of file size updates remain unchanged
    - In case of error, the folio is freed
  - No conflicts with mmap() as that writable mappings are not supported

What else ?

- Other problems ?
- Feature requests ?

Western Digital®