ORACLE

Advanced File Systems and ZFS

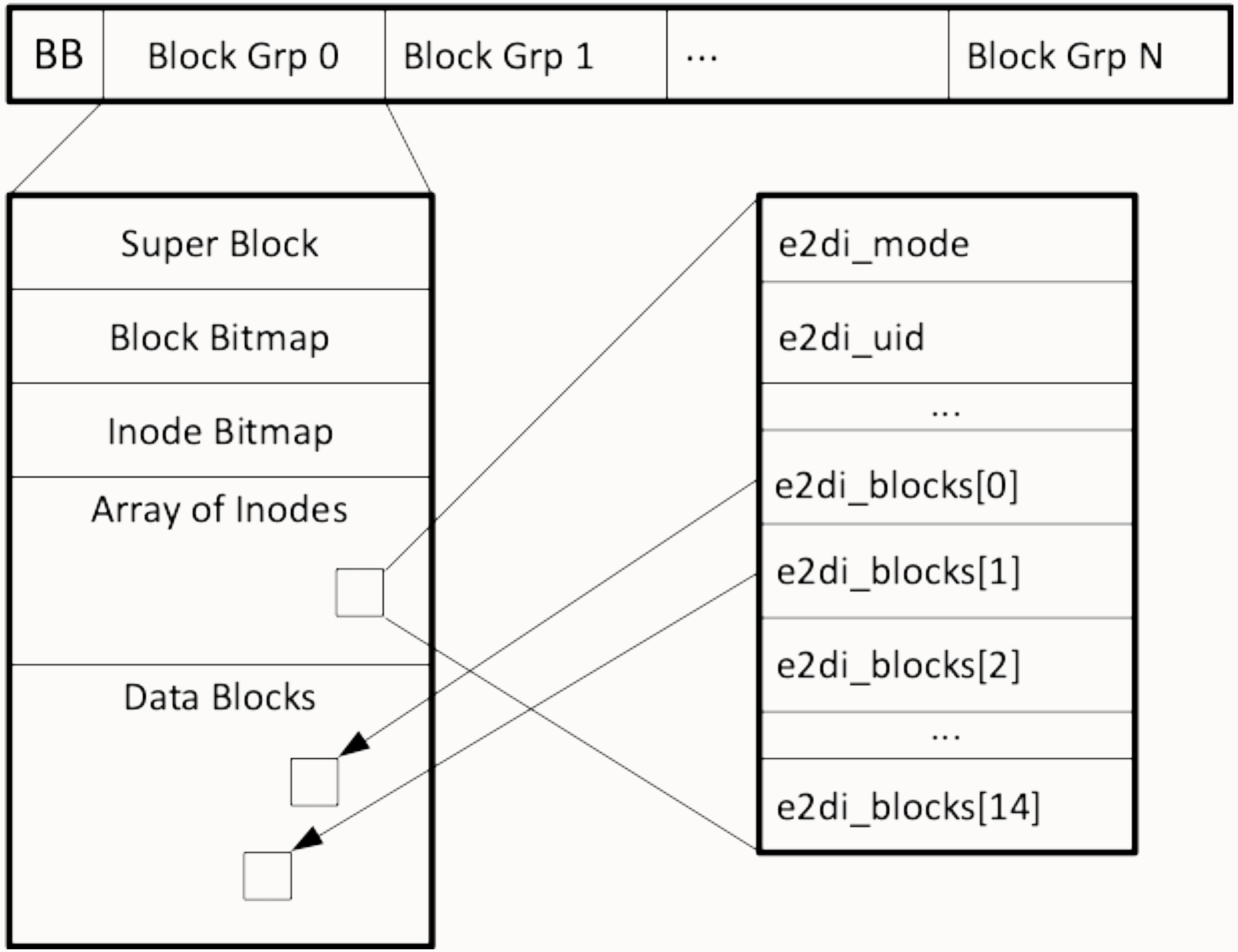
Jan Šenolt
Jan.Senolt@Oracle.COM
Solaris Engineering
May 6, 2021

Agenda

- Crash Consistency Problem
  - fsck
  - Journalling
  - Log-structured File Systems
  - Soft-Updates
- ZFS

Crash Consistency Problem

Traditional UNIX File System



- appending a new block to the file involves at least 3 writes to different data structures:
  - block bitmap - allocate the block
  - inode - update `e2di_blocks[]`, `e2di_size`
  - data block - actual payload
- what will happen if we fail to make some of these changes persistent?
  - crash-consistency problem

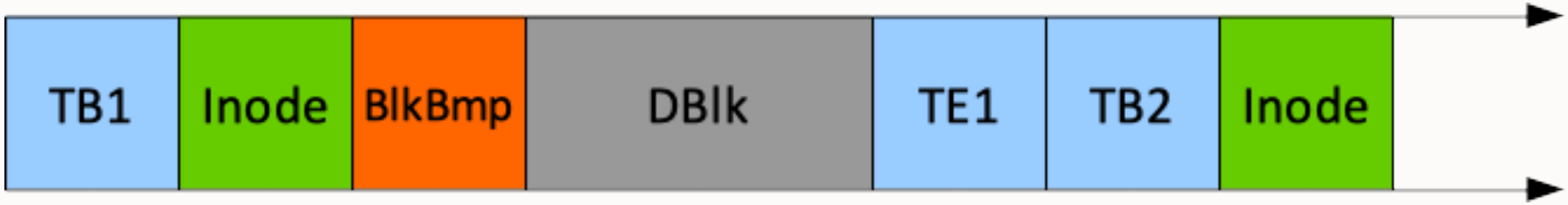
- File System Inconsistency
  - how to deal with?

File System Checker, fsck

- a reactive approach
  - let the inconsistencies happen and try to find (and eventually fix) them later (on reboot)
- metadata-only based checks
  - verify that each allocated block is referenced by exactly one inode
    - … but what if it is not??
  - unable to detect corrupted (missing) user data
- does not scale well
  - O(file system size)
- improvements?
  - check only recently changed data?
- … still useful!

Journaling, logging

1. start a new transaction
2. write all planned change to the journal
3. make sure that all writes to log completed properly
    - close the transaction
4. make the actual in-place updates



- journal reply
    - after crash, on reboot
    - walk the journal, find all complete transactions and apply them

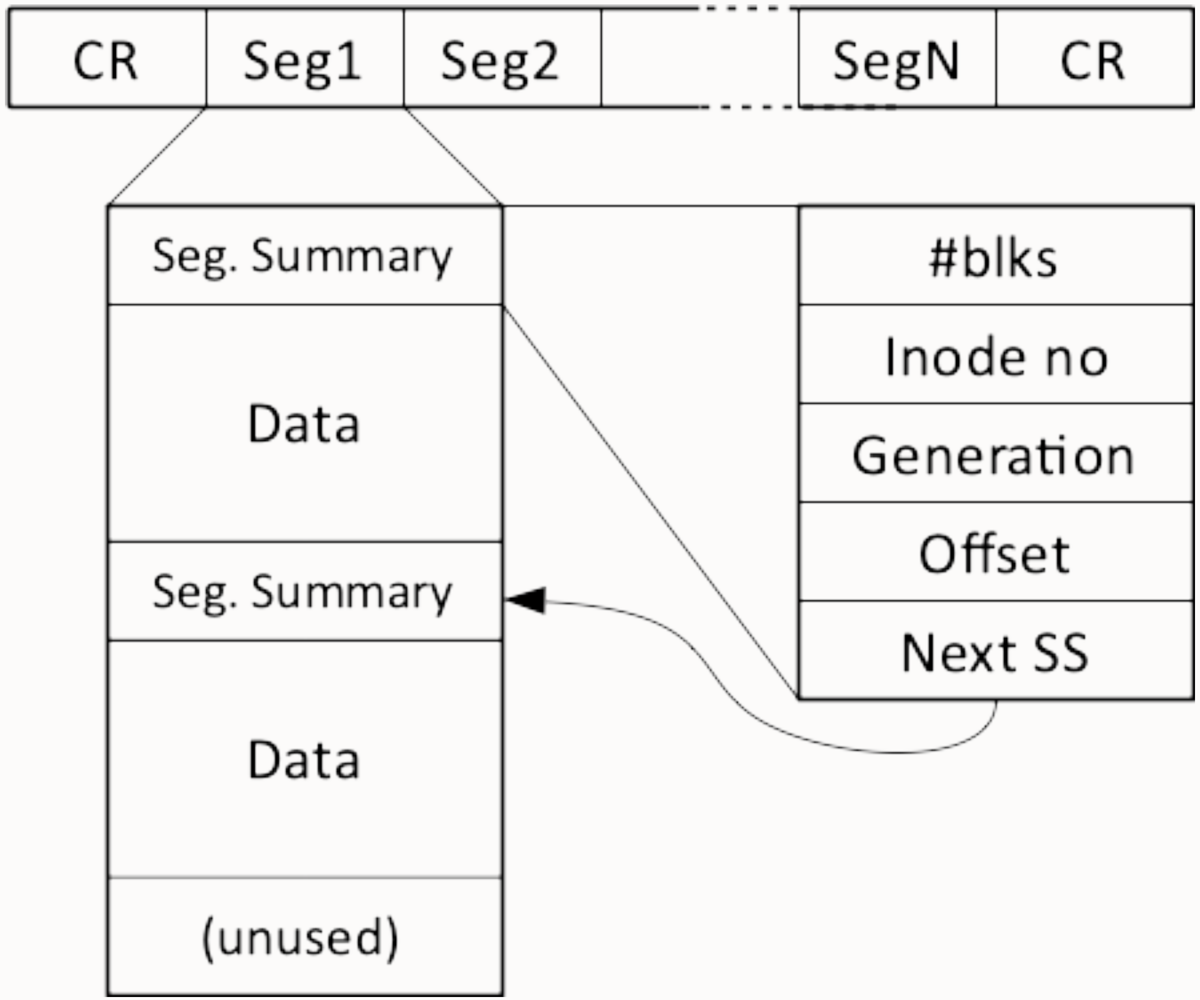
Journaling, logging (2)

- journal can be a (preallocated) file within the file system or a dedicated device
  - small circular buffer
    - UFS: 1MB per 1GB, 64MB max
- types of journals
  - physical - stores the actual content of blocks (UFS, ext2, …)
    - requires more space but it’s easy to reply
  - logical - description of the change (ZFS)
    - must be idempotent
  - redo or intent - changes to be done (UFS, ZFS, VxFS, …)
  - undo - previous content
    - undo/redo

Journaling, logging (3) - improvements

- journal aggregation
  - do multiple changes in memory, log them together in one transaction
  - efficient when updating the same data multiple times
  - longer transaction —> more data lost in case of crash
- log rolling
  - file system writes primarily the log, some other thread processes the log and performs in-place changes
- metadata-only journal
  - lower write overhead
  - how to deal with data blocks?
    - write after the transaction
      - inode can point to garbage
    - write before the transaction
      - block reuse problem

Log-structured File System



- “logging file system without the file system”
- never overwrite any data
  - write all changed data to an empty segment
  - fast crash recovery
- long sequential writes and aggressive caching
  - better I/O bandwidth utilisation
- disk has finite size
  - some sort of garbage collecting needed
- Checkpoint Regions

Log-structured File System (2)

- segment cleaner (garbage collector)
    1. read whole segment(s) into memory
    2. write all live data to another free segment(s)
        - live data - referenced by an inode
    3. mark the original segment as empty
- all live data is constantly moving around, so where is my inode?
    - inode map - inode lookup table (array)
        - kept in memory
        - stored within segments but location is stored in Checkpoint Regions
        - can be build from scratch by reading the disk content

Soft Updates

- enforces rules for data updates:
  - never point to an uninitialised structure (e.g. an inode must be initialised before a dir entry references it)
  - never reuse block which is still referenced (e.g. an inode’s pointer must be cleared before the data block may be reallocated)
  - never remove existing reference until the new one exists (e.g. do not remove the old dir entry before the new one has been written)
- keeps changed blocks in memory, maintains their update dependencies and eventually write them asynchronously
- can start using the file system immediately after the crash
  - the worst case scenario is a block leak
  - run fsck later or on background
- very complex, hard to implement properly

References

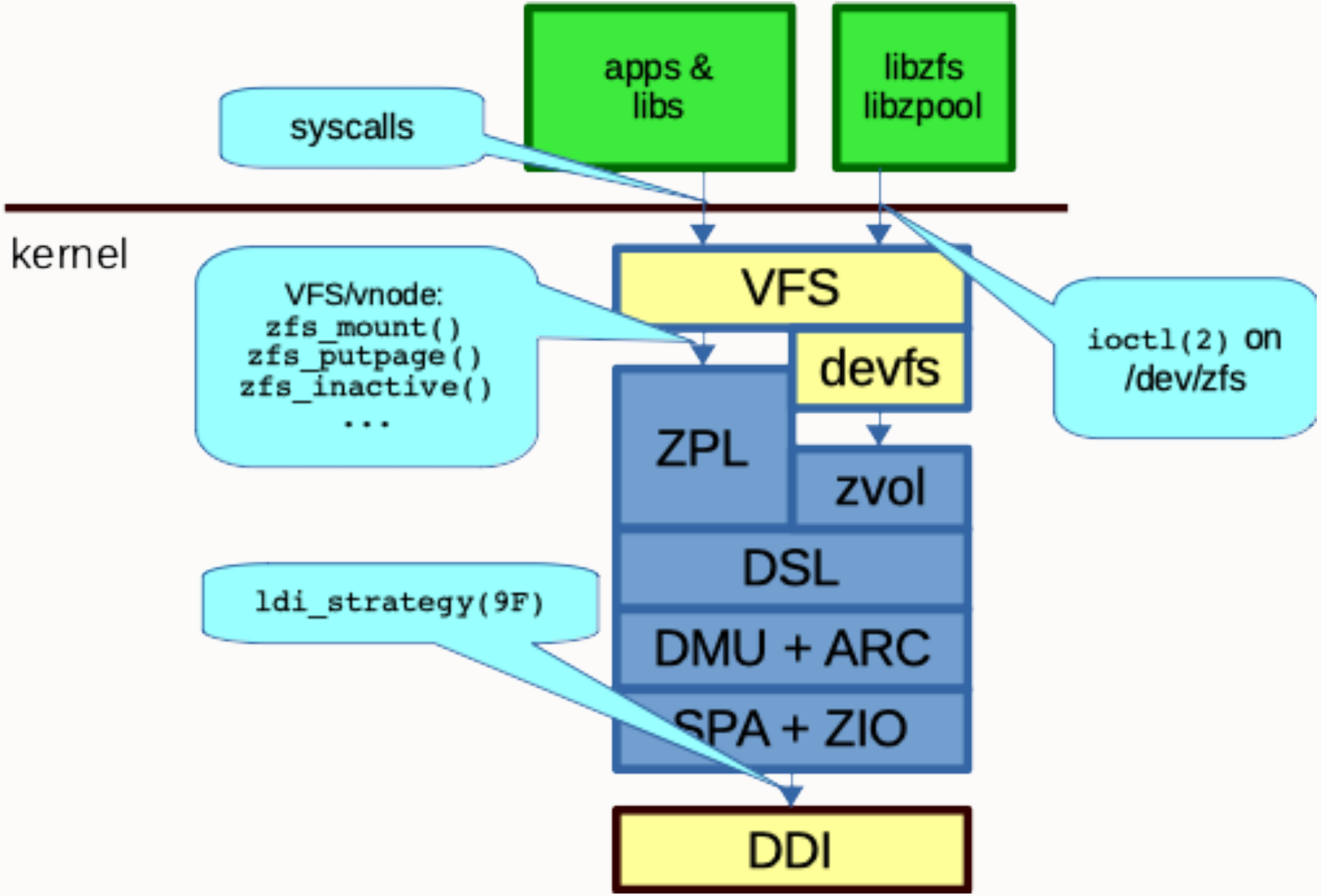
- M. K. McKusick: “Improving the Performance of fsck in FreeBSD”, ;login, 2013
- Stephen C. Tweedie: “Journaling the Linux ext2fs Filesystem”, Proceeding of the 4th Annual LinuxExpo, 1998
- M. Rosenblum, J. K. Ousterhout: “The Design and Implementation of a Log-Structured File System”, ACM Transactions, February 1992
- V. Aurora: "Soft updates, hard problems”, LWN, 2009

ZFS

ZFS vs traditional File Systems

- New administrative model
  - 2 commands: `zpool(8)` and `zfs(8)`
  - pooled storage
    - eliminates the notion of volumes, slices, …
  - dynamically allocated data structures (inodes, …)
- Integrated data protection
  - transaction-based
  - RAID 0, 1, 10, RAID-Z
  - “self-healing” (detects and corrects data corruption)
- Advanced features
  - (writable) snapshots, transparent compression, encryption, deduplication, replication, integrated NFS & CIFS sharing

ZFS in Solaris



Pooled Storage Layer, SPA

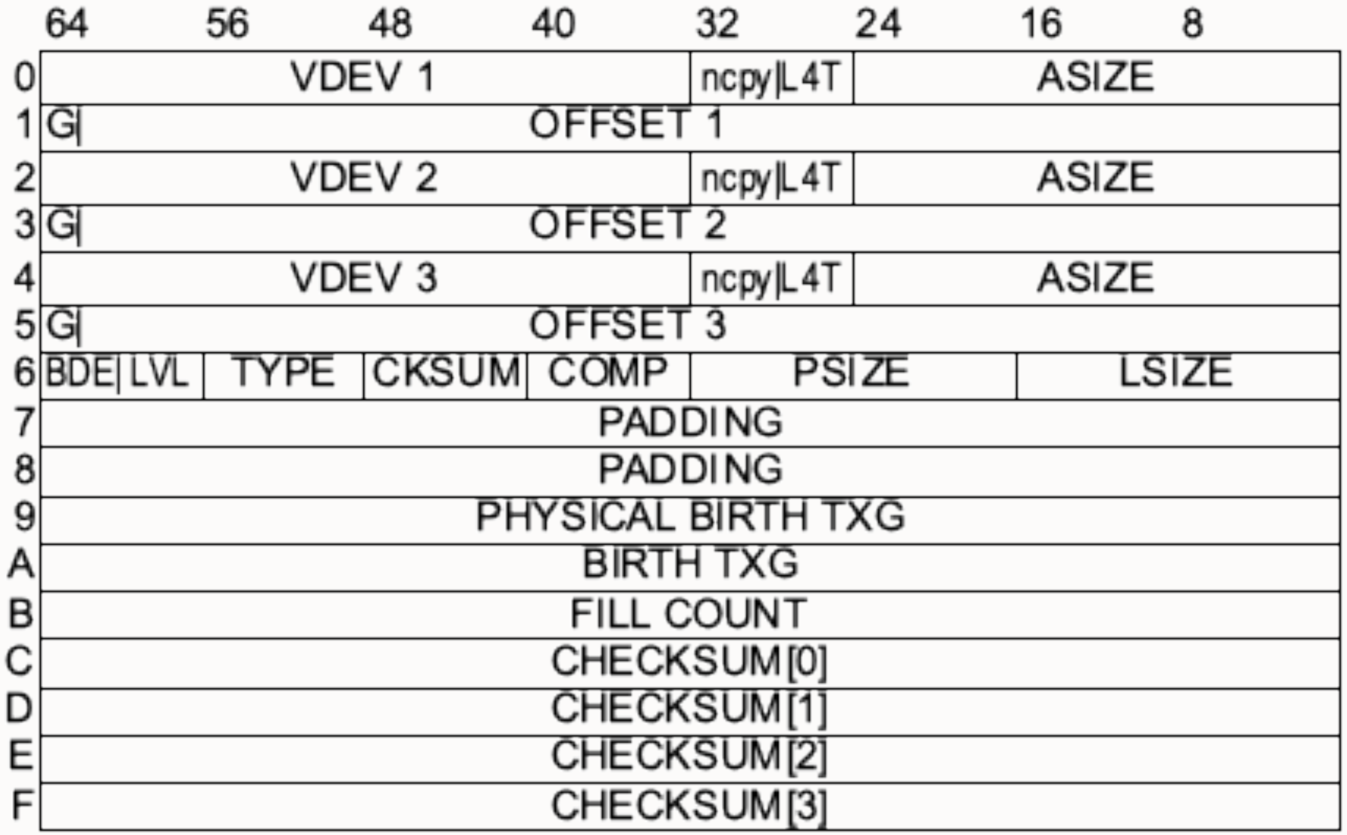
- ZFS pool
  - collection of blocks allocated within a vdev hierarchy
  - top-level vdev(s)
  - physical vdev(s)
    - leaf only
    - block device or a file
  - logical vdev
    - implements RAID
  - special vdev(s)
    - l2arc, log, meta
- ZIO
  - pipelined parallel I/O subsystem
  - performs aggregation, compression, converts endianity
  - calculates and verifies checksums (self-healing)

```
# zpool status mypool
  pool: mypool
    id: 4340326651853499056
 state: ONLINE
  scan: none requested
config:

        NAME                STATE     READ WRITE CKSUM
        mypool              ONLINE       0     0     0
          mirror-0          ONLINE       0     0     0
            c1t1d0          ONLINE       0     0     0
            c1t2d0          ONLINE       0     0     0
          /var/tmp/big_file ONLINE       0     0     0
        logs
          c1t3d0            ONLINE       0     0     0
```

Pooled Storage Layer, blkptr_t

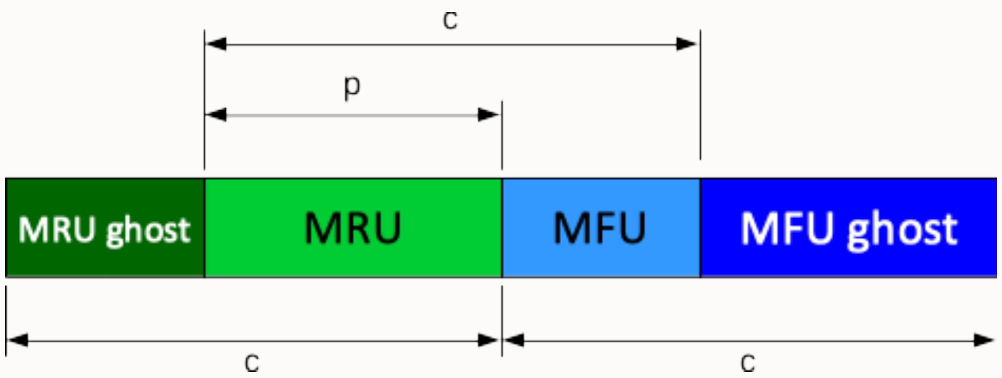
- DVA - Disk Virtual Address
    - VDEV - top-level vdev number
    - ASIZE - allocated size
- LSIZE - logical size
    - without compression, RAID-Z or gang overhead
- PSIZE - compressed size
- LVL - block level
    - 0 … data block
    - > 0 … indirect block
- FILL COUNT - number of blkptrs in block
- TYPE - type of pointed object
- BDE - endianess, deduplication, encryption



Data Management Unit, DMU

- dbuf (`dmu_buf_t`)
  - in-core data block, stored in ARC
  - 512B - 1MB
- object (`dnode_t, dnode_phys_t`)
  - array of dbufs
  - ~60 types: DMU_OT_PLAIN_FILE_CONTENTS, DMU_OT_DIRECTORY_CONTENTS,…
  - `dn_dbufs` - list of dbufs
  - `dn_dirty_records` - list of modified dbufs
- objset (`objset_t, objset_phys_t`)
  - set of objects
  - `os_dirty_dnodes` - list of modified dnodes

Adaptive Replacement Cache, ARC



- MRU - blocks seen only once recently, c is its target size
- MFU - blocks seen more than once recently, (p - c) is its target size
- `arc_adapt()`
  - p - increase if found in MRU-Ghost, decrease if found in MFU-Ghost
  - c - increase to fill available memory
- replacement policy when cache is full: if MRU size is < c, replace in MRU, else replace in MFU
- Hash table
  - hash(SPA, DVA, TXG)
  - `arc_hash_find()`, `arc_hash_insert()`
  - `arc_promote_buf()` - move from MRU to MFU

Adaptive Replacement Cache, ARC

- Unfortunately, we don’t have infinite memory
  - ARC sometimes must shrink and release memory to other consumer
  - `arc_reclaim_thread`
    - evict list - list of unreferenced dbufs —> can be removed
  - `arc_reaper_thread` (Solaris 10)
    - forces the SLAB allocator to release as many pages as possible, purge all magazines
    - very painful operation
- `arc_kill_buf()` - move a buffer to the ghost state
- L2ARC
  - persistent extension of ARC
  - `l2arc_feed_thread()` moves dbufs from ARC to L2ARC
    - `l2arc_eligible()`

Dataset and Snapshot Layer, DSL

- adds names to objsets
- creates parent - child relation
- implements snapshots and clones
- maintains properties
- DSL scan - traverses the pool, triggers self-healing
    - scrub - scans everything, like `fsck(1)`
    - resilver - scans only txgs when the vdev was missing
    - 2 phases:
        1. collect blocks to scan and sort them by offset
        2. scan blocks sequentially
- ZFS stream
    - serialised dataset(s)

ZFS POSIX Layer, ZPL & ZFS Volumes

- ZPL
  - creates a POSIX-like file system on top of DSL dataset
  - `znode_t, zfsvfs_t`
  - System Atributes (SA)
    - portion of znode with variable layout to accommodate various attributes (ACLs)
- ZVOL
  - creates a block device on top of DSL dataset
    - have entries in `/dev/zvol/[r]dsk`
  - can be shared via COMSTAR
    - iSCSI, FC target
    - direct access to DMU & ARC, Remote DMA

Write to file (1)

```
zfs_putapage(vnode, page, off, len, …):
    dmu_tx_t *tx = dmu_tx_create(vnode->zfsvfs->z_os);
    dmu_tx_hold_write(tx, vnode->zp->z_id, off, len);
    err = dmu_tx_assign(tx, TXG_NOWAIT);
    if (err)
          dmu_tx_abort(tx);
          return;
    dmu_buf_hold_array(z_os, z_id, off, len, ..., &dbp);
    bcopy(page, dbp[]->db_db_data);
    dmu_buf_rele_array(dbp,…);
    dmu_tx_commit(tx);
```

dmu_buf_t **dbp

Write to file (2), dmu_tx_hold_*

- what we are going to modify?

```
dmu_tx {
  list_t tx_holds;
  objset_t
  *tx_objset;
  int tx_txg;
  …
}
```

```
dmu_tx_hold {
  dnode_t txh_dnode;
  int txh_space_towrite;
  int txh_space_tofree;
  …
}
```

- `dmu_tx_hold_free`(), `dmu_tx_hold_bonus`(), …

Write to file (3), dmu_tx_assign()

- assign the tx to the open TXG

```
dmu_tx_try_assign(tx):
  for txh in tx->tx_holds:
    towrite += txh->txh_space_towrite;
    tofree += txh->txh_space_tofree;
  […]
  dsl_pool_tempreserve_space();
```

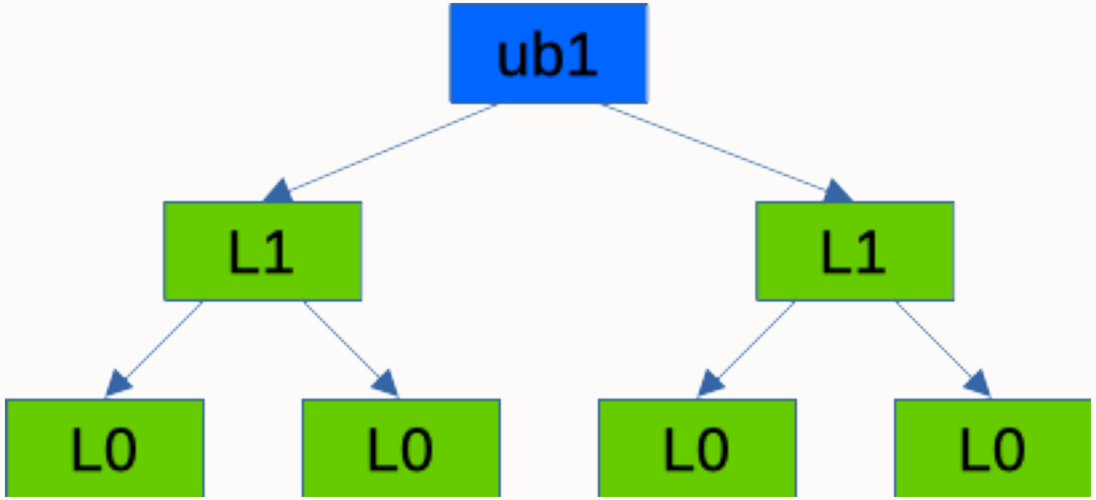
→

```
dsl_pool_tempreserve_space():
  if (towrite + used > quota)
    return (ENOSPC);
  if (towrite > arc->avail)
    return (ENOMEM);
  if (towrite > write_limit)
    return (ERESTART);
  ...
```

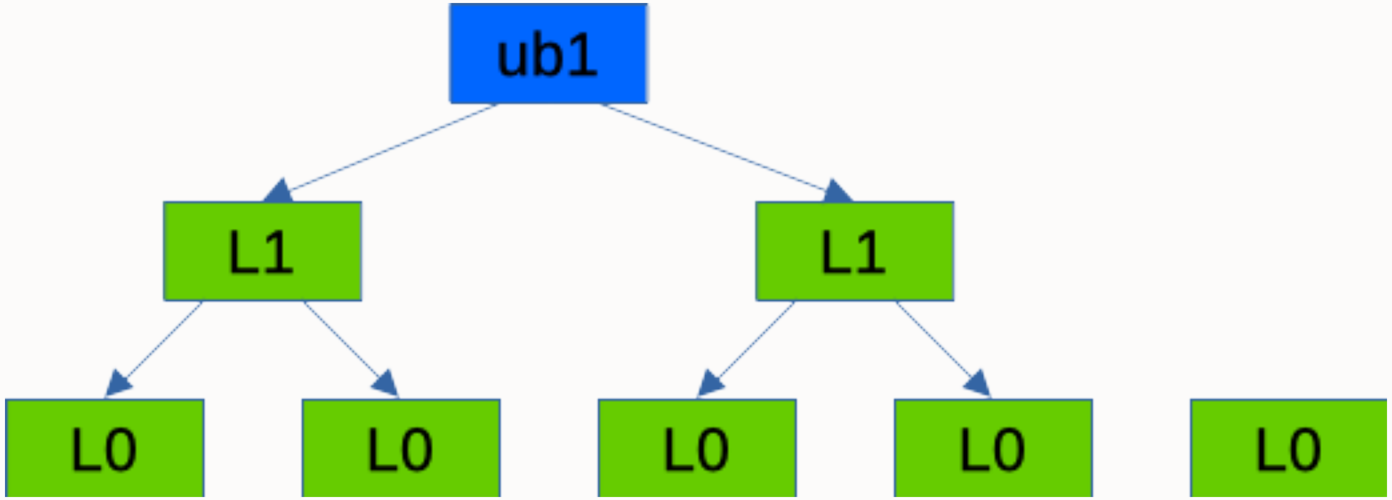
Write to file (4), TXG Life Cycle

- each TXG goes through 3-stage DMU pipeline:
  - open
    - accepts new `dmu_tx_assign()`
  - quiescing
    - waits for every TX to call `dmu_tx_commit()`
    - `txg_quiesce_thread()`
  - syncing
    - writes changes to disks
    - `txg_sync_thread()`
      - `spa_sync()`

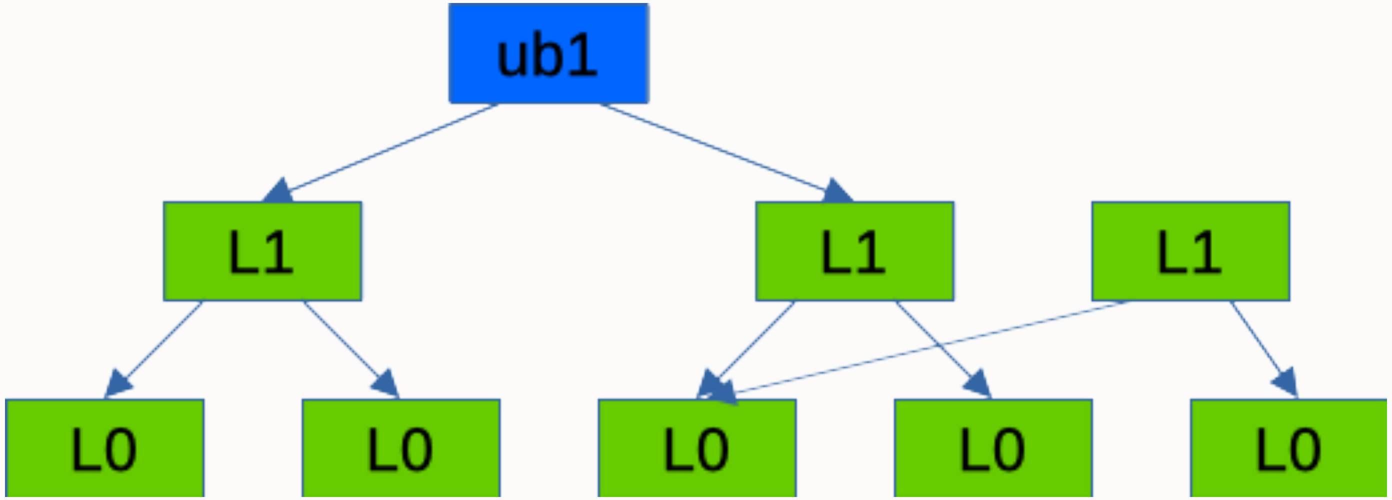
Write to file (4), Sync Phase



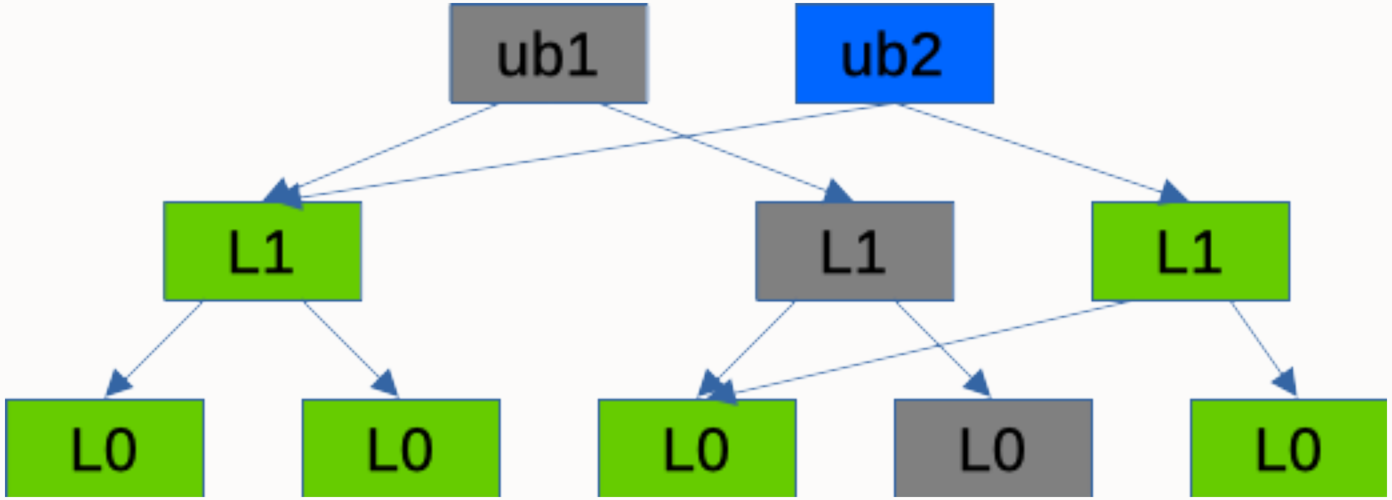
Write to file (4), Sync Phase



Write to file (4), Sync Phase



Write to file (4), Sync Phase



Write to file (5), ZIO

- depending on the IO type, dbuf properties etc ZIO goes through different stages of the ZIO pipeline:
  - ZIO_STAGE_WRITE_BP_INIT - data compression
  - ZIO_STAGE_ISSUE_ASYNC - moves ZIO processing to `taskq(9F)`
  - ZIO_STAGE_CHECKSUM_GENERATE - checksum calculation
  - ZIO_STAGE_DVA_ALLOCATE - block allocation, `metaslab_alloc_dva()`
  - ZIO_STAGE_READY - synchronisation
  - ZIO_STAGE_VDEV_IO_START - start the write by calling `vdev_op_io_start` method
  - ZIO_STAGE_VDEV_IO_DONE
  - ZIO_STAGE_VDEV_IO_ASSES - handle eventual write error

Free Space tracking

- none
  - free = not-allocated —> not necessary to track free space explicitly
  - CP/M, FAT
- bitmap
  - array of bits, each bit represents a data block.
  - for 8K block: 16K ~ 1G, 16M ~ 1TB, 16G ~ 1PB
    - slow to scan
- B-Tree of extents
  - alloc is much better
  - slow random frees
- deferred frees
  - keep list of recently freed blocks in memory

Space Allocation in ZFS (1)

- each top-level vdev is split into 200 metaslabs
  - don’t need to keep inactive metaslabs in RAM
- each meta slab has associated a space map
  - in core - AVL trees of extents, sorted:
    - by offset - easy to coalesce extents
    - by size - for searching by extent size
  - on disk - time ordered log of allocations and frees
    - append-only
    - destroy and recreate from the tree when log is too big
    - the last block is kept in ARC

Space Allocation in ZFS (2)

- space change entry

```
  1               47                   1           15
,-----------------------------------------------------------.
| 0 |   offset (sm_shift units)    | type |       run        |
`-----------------------------------------------------------'
 63  62                       17   16   15                  0
```

  - offset - offset of the extent within the metaslab (up to 64P or 512PB)
  - type - 0 = alloc
  - run - length of the extent
    - up to 16M or 128MB

- time stamp entry

```
  1      3         10                     50
,---+--------+------------+-----------------------------------.
| 1 | action |  syncpass  |        txg (lower bits)             |
`---+--------+------------+-----------------------------------'
 63  62    60 59        50 49                                 0
```

Space Allocation in ZFS (3)

```
[      0] ALLOC: txg 16182345, pass 1
[      1]    A  range: 0x100000a000-0x100000a400  size: 0x0400
[      2]    A  range: 0x1000024200-0x1000041400  size: 0x1d200
[...]
[  21219] ALLOC: txg 16182345, pass 2
[  21220]    A  range: 0x108794da00-0x1087958e00  size: 0xb400
[  21221]    A  range: 0x126cd48c00-0x126cd59400  size: 0x10800
[...]
[  21224] FREE: txg 16182345, pass 2
[  21225]    F  range: 0x101e894c00-0x101e8a6000  size: 0x11400
[  21226]    F  range: 0x10165c5600-0x10165c6200  size: 0x0c00
[...]
[  21272] ALLOC: txg 16182345, pass 3
[  21273]    A  range: 0x1087958e00-0x1087959600  size: 0x0800
[  21274]    A  range: 0x1142c29a00-0x1142c29c00  size: 0x0200
[  21275] ALLOC: txg 16182345, pass 4
[  21276]    A  range: 0x1087959600-0x108795a400  size: 0x0e00
[  21277]    A  range: 0x101db25e00-0x101db29e00  size: 0x4000
[  21278] ALLOC: txg 16182345, pass 5
[  21279]    A  range: 0x101db29e00-0x101db49e00  size: 0x20000
```

Space Allocation in ZFS (4)

- several different approaches over time
  - `metaslab_ff_alloc`
    - First Fit, with cursor for different block sizes
    - block size aligned offsets
    - sequential walk for more full metaslabs
  - `metaslab_df_alloc`
    - do First Fit for up to 70% (96%) full metaslabs, then do Best Fit
    - added 2nd AVL tree sorted by size
  - `“clump” allocator`
    - tries to find regions of multiple of requested size, expects more allocations of the same size to follow

Space Allocation in ZFS (5) - Free Space Fragmentation

- gang block
  - build a larger block from smaller ones
  - gang header
    - array of blkptrs to leaf blocks
  - adds 2 new ZIO stages
    - ZIO_STAGE_GANG_ASSEMBLE
    - ZIO_STAGE_GANG_ISSUE

- is log always better than a bitmap?
  - worst case scenario: 1G metaslab with 4K blocks
  - needs 1MB of log entries
  - only 32KB of bitmap

Q&A

Thank you!

ORACLE