An Introduction to the Implementation of ZFS

Brought to you by

Dr. Marshall Kirk McKusick

BSD Canada Conference 2015
June 13, 2015

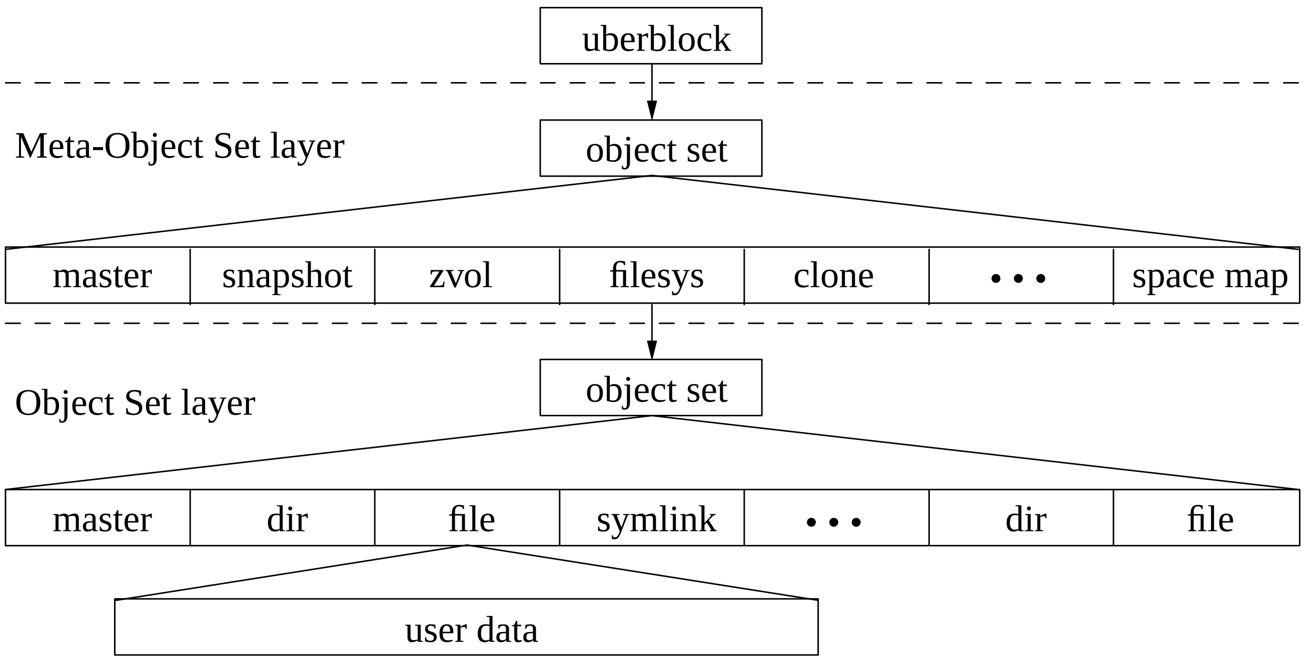
University of Ottawa
Ottawa, Canada

Zettabyte Filesystem Overview

- Never over-write an existing block
- Filesystem is always consistent
- State atomically advances at checkpoints
- Snapshots (read-only) and clones (read-write) are cheap and plentiful
- Metadata redundancy and data checksums
- Selective data compression and deduplication
- Pooled storage shared among filesystems
- Mirroring and single, double, and triple parity RAIDZ
- Space management with quotas and reservations
- Fast remote replication and backups

text ref: pp. 523-526

Structural Organization



- Uberblock anchors the pool
- Meta-object-set (MOS) describes array of filesystems, clones, snapshots, and ZVOLs
- Each MOS object references an object-set that describes its objects
- Filesystem object sets describe an array of files, directories, etc.
- Each filesystem object describes an array of bytes

text ref: pp. 527-528

ZFS Block Pointer

<table>
<tr><th></th><th colspan="8">64</th><th colspan="8">56</th><th colspan="8">48</th><th colspan="8">40</th><th colspan="8">32</th><th colspan="8">24</th><th colspan="8">16</th><th colspan="8">8</th><th>0</th></tr>
<tr><td>0</td><td colspan="32">vdev1</td><td colspan="8">grid</td><td colspan="24">asize</td></tr>
<tr><td>1</td><td colspan="1">G</td><td colspan="63">offset1</td></tr>
<tr><td>2</td><td colspan="32">vdev2</td><td colspan="8">grid</td><td colspan="24">asize</td></tr>
<tr><td>3</td><td colspan="1">G</td><td colspan="63">offset2</td></tr>
<tr><td>4</td><td colspan="32">vdev3</td><td colspan="8">grid</td><td colspan="24">asize</td></tr>
<tr><td>5</td><td colspan="1">G</td><td colspan="63">offset3</td></tr>
<tr><td>6</td><td>B</td><td>D</td><td>X</td><td colspan="5">lvl</td><td colspan="8">type</td><td colspan="8">cksum</td><td colspan="8">comp</td><td colspan="16">psize</td><td colspan="16">lsize</td></tr>
<tr><td>7</td><td colspan="64">spare</td></tr>
<tr><td>8</td><td colspan="64">spare</td></tr>
<tr><td>9</td><td colspan="64">physical birth time</td></tr>
<tr><td>A</td><td colspan="64">logical birth time</td></tr>
<tr><td>B</td><td colspan="64">fill count</td></tr>
<tr><td>C</td><td colspan="64">checksum[0]</td></tr>
<tr><td>D</td><td colspan="64">checksum[1]</td></tr>
<tr><td>E</td><td colspan="64">checksum[2]</td></tr>
<tr><td>F</td><td colspan="64">checksum[3]</td></tr>
</table>

- Up to three levels of redundancy
- Checksum separate from data
- Birth time is the transaction-group number in which it was allocated
- Maintains allocated, physical (compressed), and logical sizes

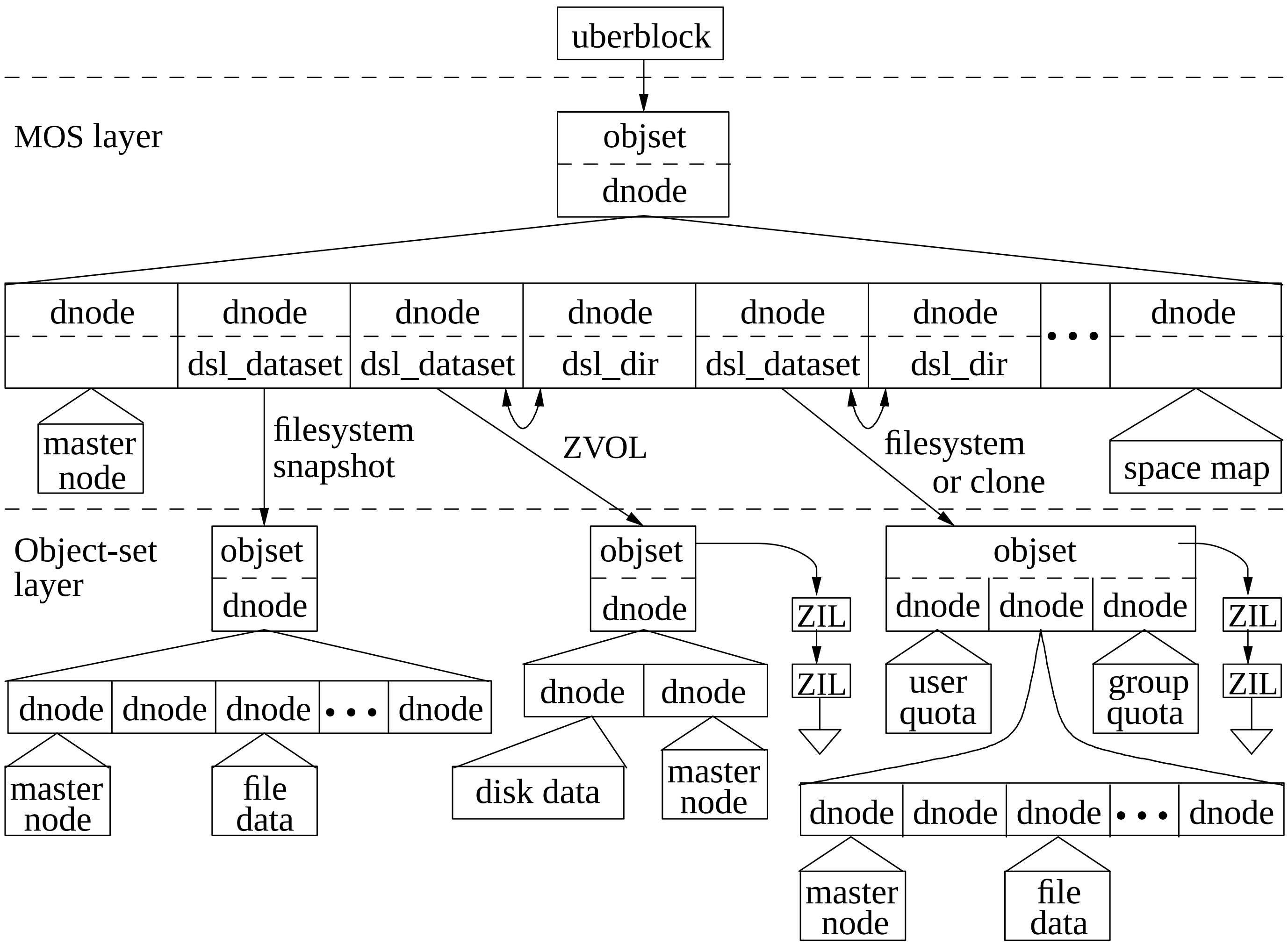
text ref: pp. 529-531

ZFS Block Management

- Disk blocks are kept in a pool
- Multiple filesystems and their snapshots are held in the pool
- Blocks from the pool are given to filesystems on demand and reclaimed to the pool when freed
- Space may be reserved to ensure future availability
- Quotas may be imposed to limit the space that may be used

text ref: pp. 542-545

ZFS Structure



- MOS layer manages space and objects using that space
- Object-set layer manages filesystems, snapshots, clones, and ZVOLs

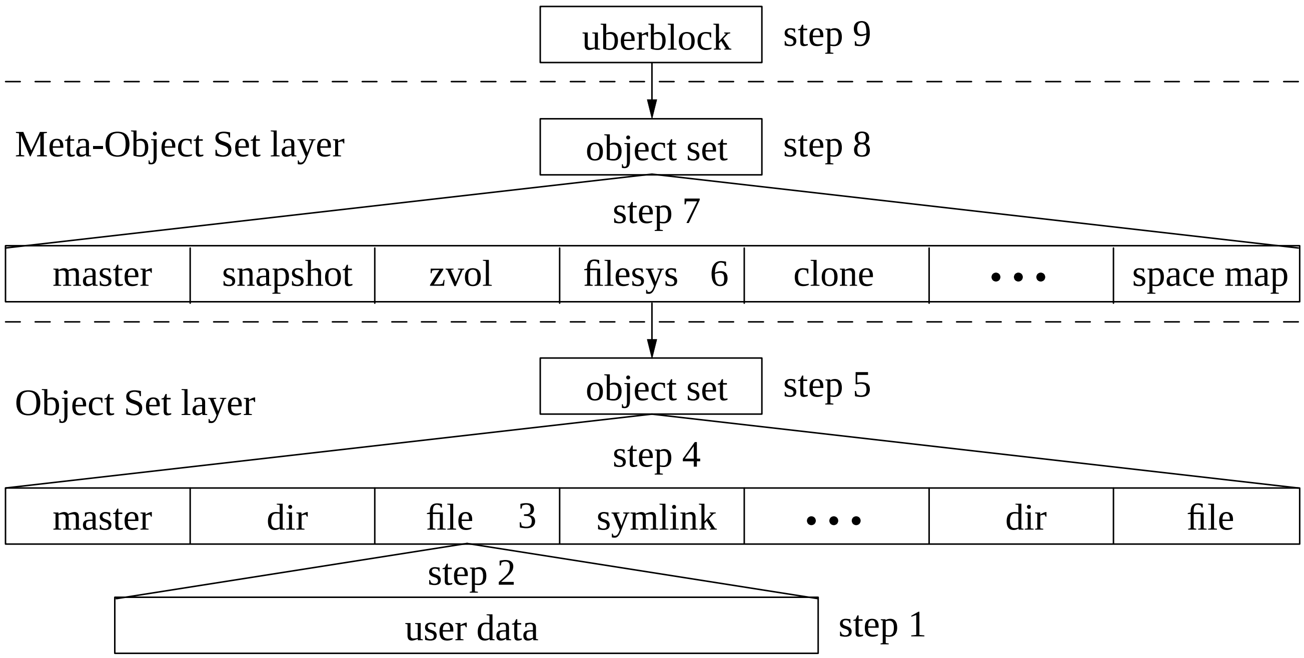
text ref: pp. 532-535

ZFS Checkpoint

- Collect all updates in memory
- Periodically write all changes to an unused location to create a checkpoint
- Last step in checkpoint writes a new uberblock
- Entire pool is always consistent
- Checkpoint affects all filesystems, clones, snapshots, and ZVOL in the pool
- Need to log any changes between checkpoints that need to be persistent
- The **fsync** system call is implemented by forcing a log write not by doing a checkpoint
- Recovery starts from last checkpoint, rolls forward through log, then creates new checkpoint

text ref: pp. 535-536

Flushing Dirty Data



Write modified data in this order:

1) new or modified user data

2) indirect blocks to new user data

3) new dnode block

4) indirect blocks to modified dnodes

5) object-set dnode for filesystem dnodes

6) filesystem dnode to reference objset dnode

7) indirect blocks to modified meta-objects

8) MOS object-set for meta-object dnode

9) new uberblock (plus its copies)

text ref: pp. 536-538

RAIDZ

	disk 1	disk2	disk 3	disk 4	disk 5
stripe 1	P0	D0	D2	D4	D6
stripe 2	P1	D1	D3	D5	D7
stripe 3	P0	D0	D1	D2	P0
stripe 4	D0	D1	D2	P0	D0
stripe 5	P0	D0	D4	D8	D11
stripe 6	P1	D1	D5	D9	D12
stripe 7	P2	D2	D6	D10	D13
stripe 8	P3	D3	D7	P0	D0
stripe 9	D1	D2	D3	X	P0
stripe 10	D0	D1	X	P0	D0
stripe 11	D3	D6	D9	P1	D1
stripe 12	D4	D7	D10	P2	D2
stripe 13	D5	D8			

- Variable size stripes since each block knows its size
- Parity sectors on disk that starts block
- N == 1, up to four blocks per parity sector.
- Blocks must be multiple of N + 1 sectors

RAIDZ Recovery

- Rebuild by traversing all MOS objects and rebuild their blocks

  Fast when pool not fully allocated as only used blocks are rewritten

  Slow when pool is full as many random reads needed

  use physical birth time to determine blocks that need to be rebuilt

- Never need to recalculate and write parity

text ref: p. 541

Freeing Filesystem Blocks

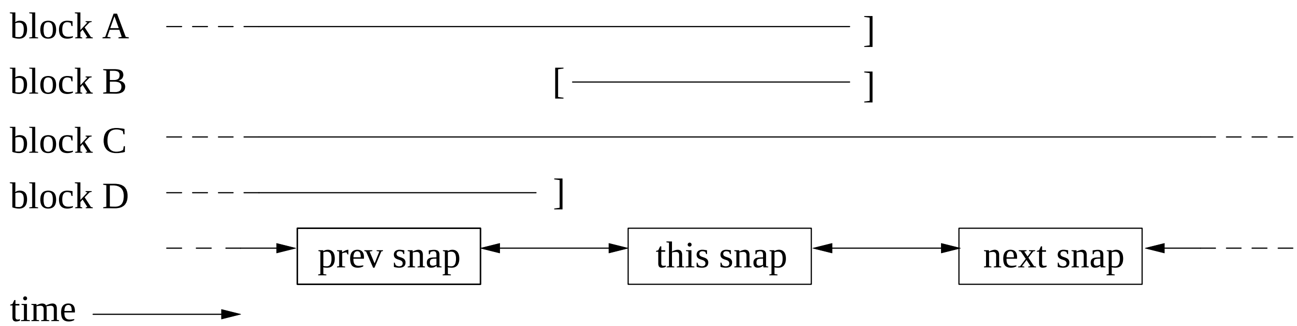
- Blocks are tracked using space maps, birth time, and deadlists
- When a block is allocated, its birth time is set to the current transaction number
- Over time, snapshots are taken which also reference the block
- When a file is overwritten, truncated, or deleted, its blocks are released
- For each freed block, the kernel must determine if a snapshot still references it

    if born after most recent snapshot, it can be freed

    otherwise it is added to the filesystem's deadlist

text ref: pp. 543-545

Freeing Snapshot Blocks



- Freeing “this snap”

  Iterate over “next snap” deadlist (blocks A and B)

  Block A born before “prev snap” so added to “this snap” deadlist

  Block B born after “prev snap” so must be freed

  Move deadlist of “this snap” to become deadlist of “next snap”

  Remove “this snap” from list of snapshots and directory of snapshot names

- Never need to make a pass over entire block-allocation map

ZFS Strengths

- High write throughput
- Fast RAIDZ reconstruction on pools with less than 40% utilization
- Avoids RAID "write hole"
- Blocks move between filesystems as needed
- Integration eases administration (mount points, exports, etc)

text ref: p. 547

ZFS Weaknesses

- Slowly written files scattered on disk
- Slow RAIDZ reconstruction on pools with greater than 40% utilization
- Block cache must fit in the kernel’s address space, thus works well only on 64-bit systems
- Needs under 75% utilization for good write performance
- RAIDZ has high overhead for small block sizes such as 4 Kbyte blocks typically used by databases and ZVOLs.
- Blocks cached in memory are not part of the unified memory cache so inefficient for files and executables using **mmap** or when using **sendfile**

text ref: pp. 548-549

Questions

More on ZFS:

- ‘‘The Design and Implementation of the FreeBSD Operating System, 2nd Edition’’, Chapter 10
- Manual pages: zfs(8), zpool(8), zdb(8)

Marshall Kirk McKusick

<mckusick@mckusick.com>

http://www.mckusick.com