Storage Systems

NPTEL Course

Jan 2012

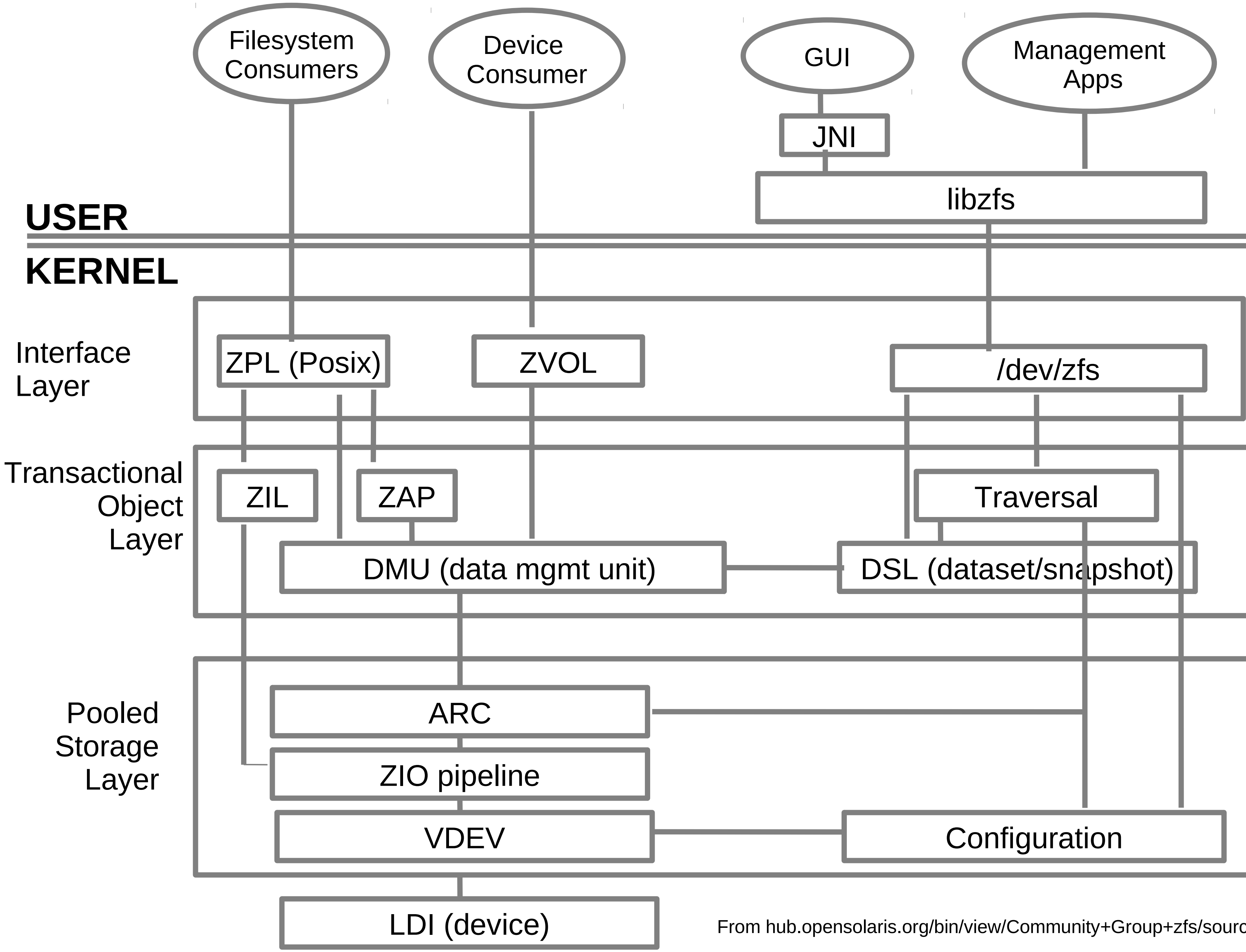
(Lecture 25)

K. Gopinath

Indian Institute of Science

Design

- User level:
  - FS consumer: uses Posix ZFS fs
  - device consumer: uses devices avlbl thru /dev
  - GUI (JNI), Mgmt Apps (both access ker thru libzfs)
    - eg. zpool(1M), zfs(1M)
    - libzfs: unified, object-based mechanism for accessing and manipulating storage pools and filesystems
- Kernel Level:
  - Interface Layer
  - Transactional Object layer
  - Storage Pool Layer



From hub.opensolaris.org/bin/view/Community+Group+zfs/source

- ZVOL (ZFS Emulated Volume): presents raw devices backed by space from a storage pool
- DMU (Data Management Unit):
  - presents a transactional object model using flat address space presented by the SPA.
  - Consumers interact with DMU via collections of objects (objsets), objects and transactions
    - Object: an arbitrary piece of storage from SPA
    - Transaction: a set of ops that must be committed to disk as a group
- DSL (Dataset and Snapshot Layer):
  - groups DMU objsets into a hierarchical namespace, with inherited properties, as well as quota and reservation enforcement
  - manages snapshots and clones of objsets

- ZAP (ZFS Attribute Processor): uses scalable hash algorithms to create arbitrary (name, object) associations within an objset.
  - used to implement directories within the ZPL
  - used in DSL to store pool-wide properties
  - runs on top of DMU
- ZIL (ZFS Intent Log): For O_DSYNC, uses efficient per-dataset transaction log that can be replayed in event of a crash
- Traversal:
  - a safe, efficient, restartable method of walking all data within a live pool
  - basis of resilvering and scrubbing
  - walks all metadata looking for blocks modified within a certain period of time
  - due to COW, can quickly exclude large subtrees that have not been touched during an outage period

- SPA: glues ZIO and vdev layers into a consistent pool object
- ARC: Adaptive Replacement Cache
- ZIO (ZFS I/O Pipeline): all data must pass thru this multi-stage pipeline when going to or from the disk
  - translates DVAs (Device Virtual Addresses) into logical locations on a vdev
  - checksum and compression if necessary
  - splits large block into “gang of blocks”
- VDEV: Virtual devices form a tree, with a single root vdev and multiple interior (mirror and RAID-Z) and leaf (disk and file) vdevs
- LDI (Layered Driver Interface): interact with physical devices and files (VFS interfaces)

I/O types	*ZIO state*	*Compression*	*Checksum*	*Gang Blocks*	*DVA management*	*Vdev I/O*
RWFCI	open					
RWFCI	wait for children ready					
-W-		write compress				
-W-			checksum gen			
-WFC-				gang pipeline		
-WFC-				get gang header		
-W-				rewrite gang header		
--F--				free gang members		
-C-				claim gang members		
-W-					DVA allocate	
--F--					DVA free	
-C-					DVA claim	
-W-			gang checksum gen			
RWFCI	ready					
RW--I						I/O start
RW--I						I/O done
RW--I						I/O assess
RWFCI	wait for children done					
R--			checksum verify			
R--				read gang members		
R--		read decompress				
RWFCI	done					

(R)ead, (W)rite, (F)ree, (C)laim, and (I)octl (from ZFS doc: Sun/Oracle)

Types of Disk Redundancy

- Maximum Distance Separable (MDS) vs non-MDS
- MDS: RAID1, RAID4, RAID5, RAID6 popular
  - RAID1: mirroring
  - RAID5: parity rotated but not in RAID4
  - RAID m+n where RAID m used as leaves and RAID n on top
    - RAID10: create multiple RAID1 (mirroring) volumes and then catenate them in RAID 0
  - RAID6: need two syndromes for tolerating 2 failures
    - 1st one the regular RAID5: xor of $D_0, \ldots D_i, \ldots, D_{n-1}$
    - 2nd one: xor of $g^0 D_0, \ldots, g^i D_i, \ldots, g^{n-1} D_{n-1}$, g a generator in a GF
    - If one disk fails, RAID5 syndrome sufficient
    - If 2 data disks fail, say i^{th} and j^{th}: have to solve $D_i + D_j = A$ and $g^i D_i + g^j D_j = B$

RAID5 “write-hole”

- Software RAID5 perf poor
  - if data upd in a RAID stripe, must also upd parity
    - If data part upd but crash/power outage before parity upd, xor invariant of RAID stripes lost
  - a full-stripe write can issue all writes asynch, but a partial-stripe write must do synch reads before it can start the writes
- Also, a partial-stripe write modifies live data
  - defeats transactional design
- software-only workarounds: logging but slow!
- HW workaround:
  - NVRAM for both probs but costly...