Filesystems

Timothy Roscoe, David Cock

Fall 2016

https://www.systems.ethz.ch/courses/fall2016/aos

Overview

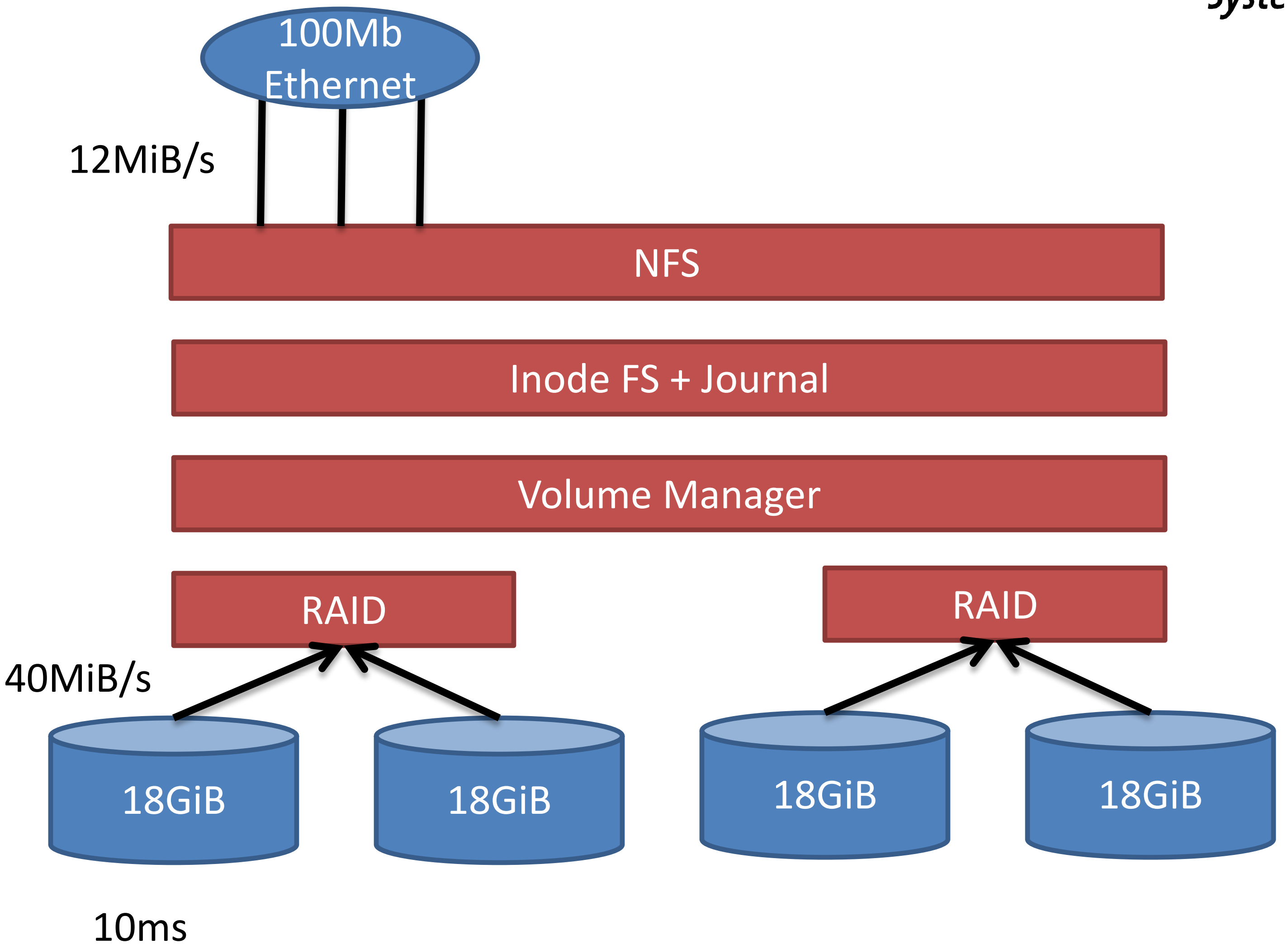
Systems@ETH Zürich

- There's been lots of FS development in 15yrs
- Much isn't (yet) in the textbook
- We'll cover a few interesting things:
  - New hardware e.g., Flash
  - Data resiliency
  - API evolution: transactionality & storage pools
- This is **far** from comprehensive
  - ZFS as an *example* of more modern design
  - BTRFS is similar

Systems@ETH Zürich

FILESYSTEMS IN 1999

A File Server



Midrange Fileserver, Late 90s

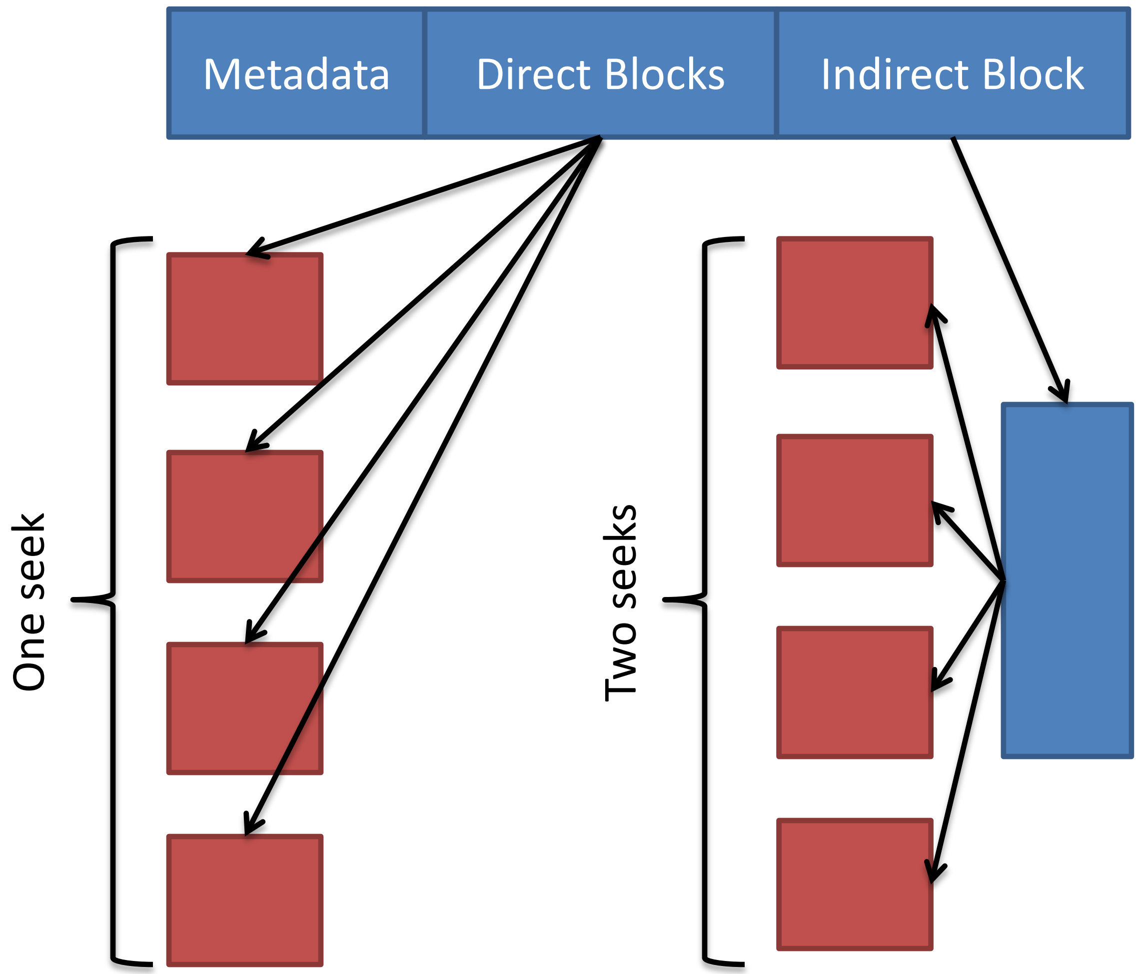
Sun Ultra Enterprise 450

- 4 x 400MHz UltraSparc II
- 4GiB RAM
- 4 x 18GiB 10kRPM UltraSCSI (72GiB)
  - Latency: **8ms**
  - BW: 4 x 35MiB/s = 140MiB/s
- 100MB Ethernet, 12MiB/s
- List price: USD34,995 (USD48,000 today)

- Seek time dominates performance, networks are slow.
- 4 drives @ 99.9% reliability = 99.6% overall reliability
- RAID rebuild takes about 10 minutes.
- Error rate: 1 in 10^{15} bits, 1/50,000 errors during rebuild

INode-Structured FS

ext+, ufs, NTFS, …



- Optimised for HDDs
- Reduces seek chains
- Improves locality with **block groups**
- 512B blocks

Atomicity and fsck

- Atomic disk operation is the **block write**.
- We could crash between two writes:
  - Removing a file, and freeing its I nodes
  - Allocating new blocks and writing them
  - Just about any non-trivial operation, actually!
- If you crash, you **fsck** – walk the disk and look for orphaned blocks.
- We can do better…

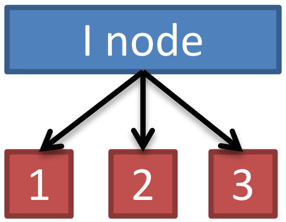
Journaling

ext3+, XFS, …

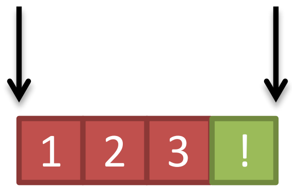
Filesystem transactions:

- Complete or rollback.
- Write the journal **first**.
- Once complete, remove the journal entry.
- If we crash, replay the journals.
- Writes every block **twice** – thus usually only journal metadata.
- The log is written sequentially, **fast**!
- *Log-structured* FS in the limit.

FS Data Structure



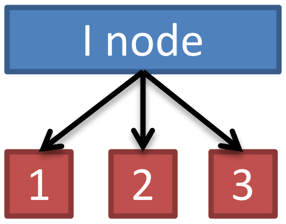
Journal



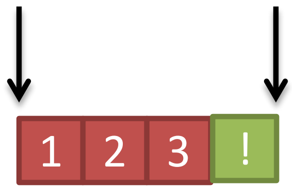
Journal Limitations

- Metadata journal only protects metadata (obviously).
- Write amplification.
- All updates must be *idempotent*
  - We may crash replaying the journal
  - The same update will happen twice

FS Data Structure

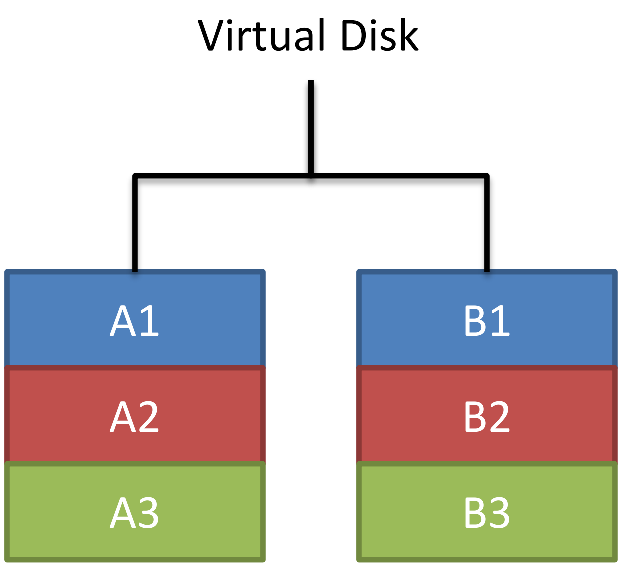


Journal



RAID

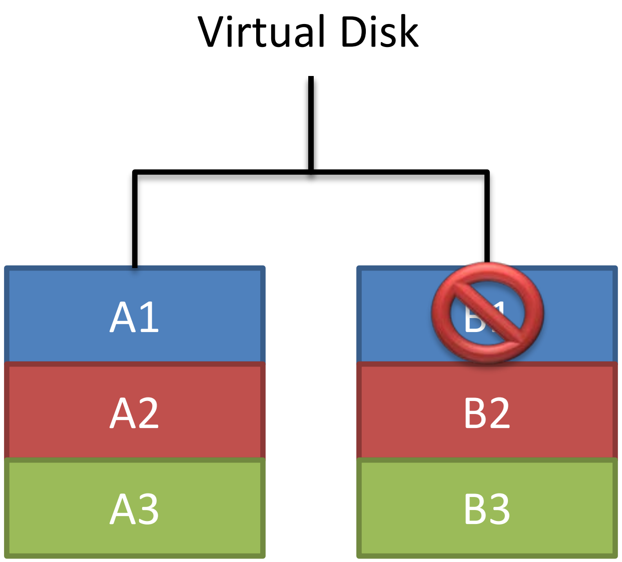
- Hard drives fail
- Distribute blocks across several drives:
  - Stripe
  - Mirror
  - ECC
- FS sees *virtual disks*



RAID 1 (mirror)

RAID Limitations

- Performance during rebuild is terrible.
- Stripe writes must be **atomic**:
  - *The RAID write hole.*
  - Proper controllers need battery-backed RAM.
- Dynamic resizing is hard



Volumes

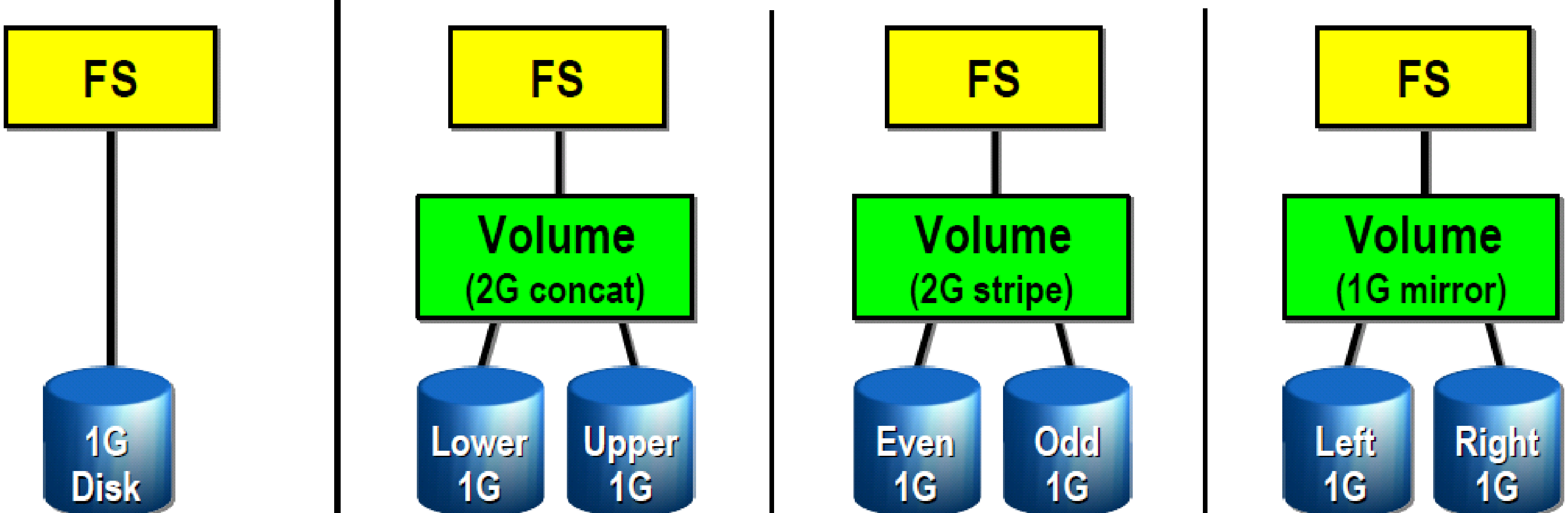
- Don't want to be limited by physical sizes
  - Increase filesystem capacity by adding new drives, without deleting and recreating it.
- View all disks as a *pool* of blocks – allocate virtual blocks to physical blocks in a (hopefully) sensible fashion
  - LVM on Linux
- May or may not integrate with RAID

Why Volumes Exist

In the beginning, each filesystem managed a single disk.

It wasn't very big.

- Customers wanted more space, bandwidth, reliability
  - Hard: redesign filesystems to solve these problems well
  - Easy: insert a shim ("volume") to cobble disks together
- An industry grew up around the FS/volume model
  - Filesystem, volume manager sold as separate products
  - Inherent problems in FS/volume interface can't be fixed



WHAT'S NEW IN 2015?

Midrange Fileserver, Late 90s

Sun Ultra Enterprise 450

- 4 x 400MHz UltraSparc II
- 4GiB RAM
- 4 x 18GiB 10kRPM UltraSCSI (72GiB)
  - Latency: **8ms**
  - BW: 4 x 35MiB/s = 140MiB/s
- 100MB Ethernet, 12MiB/s
- List price: USD34,995 (USD48,000 today)

- Seek time dominates performance, networks are slow.
- 4 drives @ 99.9% reliability = 99.6% overall reliability
- RAID rebuild takes about 10 minutes.
- Error rate: 1 in 10^{15} bits, 1/50,000 errors during rebuild

Midrange Fileserver 2015

Dell 730xd + MD3060e

- 2 x Intel 2.3GHz E5-2670 (24C/48T)
- 64GiB RAM
- 54 x 1.2TiB 10kRPM 6G SAS HDD (64.8TiB)
    - Latency: **7ms**
    - BW: 54 * 117 = 6.2GiB/s
- 40GB Ethernet, 5GiB/s
- List price: USD48,000

- 100x more CPU
- 10x more RAM
- 1000x more HDD capacity
- 50x more HDD bandwidth
- 400x more network bandwidth
- **Same latency!**

- Seek still dominates
- 54 drives @ 99.9% = 94.7% overall
- RAID rebuild takes 3.5hrs
- 1 in 10^{15} SER, 1/750 rebuild errors!

Volume Management is Hard

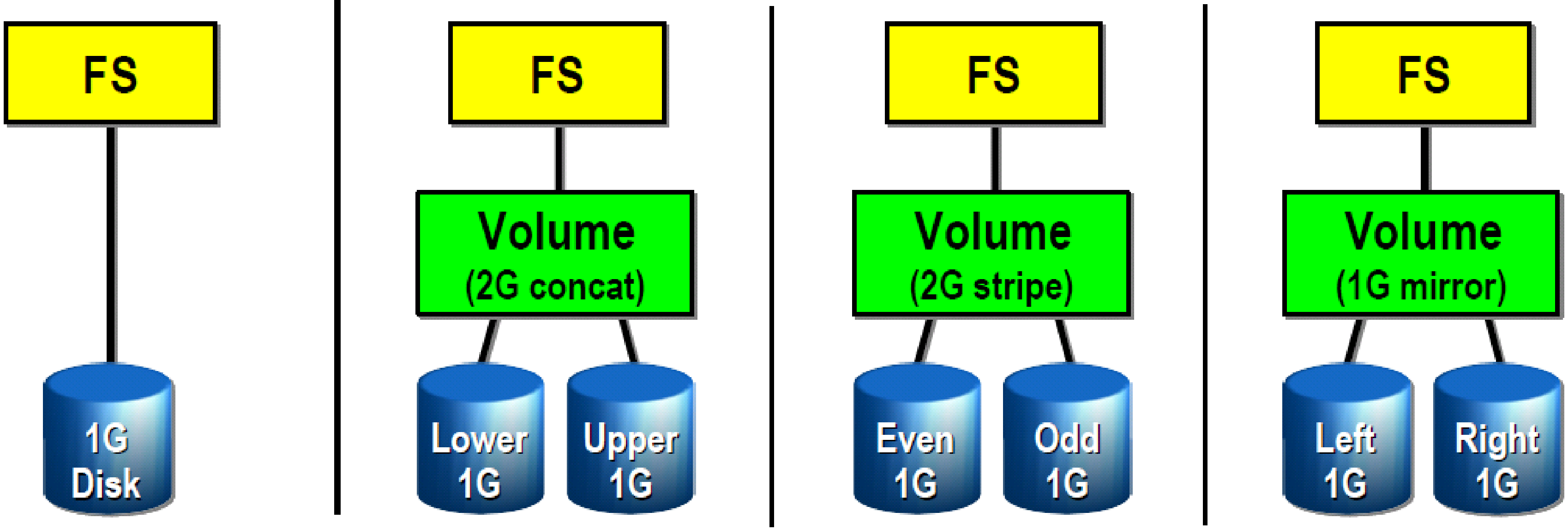
- Ever tried (hot) plugging a new drive, and resizing your root FS?
- With 4 disks, adding or removing one is rare.
- With >50, it's a weekly ritual, with 10,000 it's non-stop.
- Filesystem resize (e.g. resize2fs) is a band-aid – the FS **really** wants a physical disk.
- Virtualisation and SAN make this harder.

Why Volumes Exist

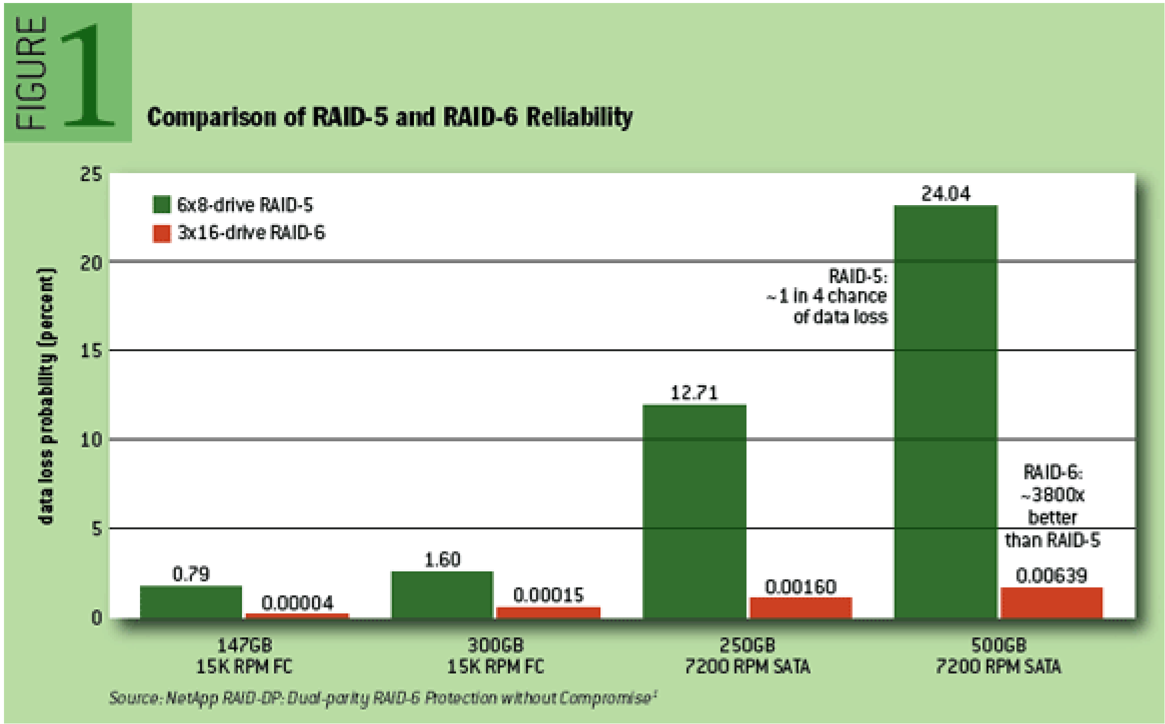
In the beginning, each filesystem managed a single disk.

It wasn't very big.

- Customers wanted more space, bandwidth, reliability
  - Hard: redesign filesystems to solve these problems well
  - Easy: insert a shim ("volume") to cobble disks together
- An industry grew up around the FS/volume model
  - Filesystem, volume manager sold as separate products
  - Inherent problems in FS/volume interface can't be fixed



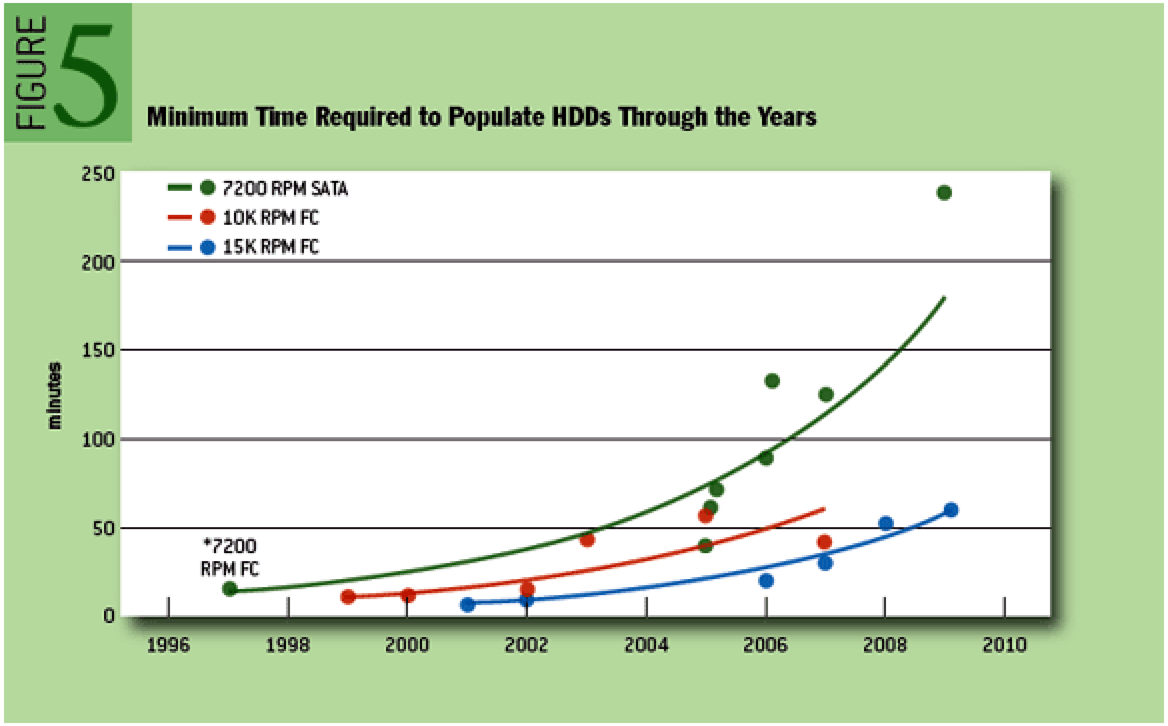
Redundancy is Hard



Adam Leventhal, Triple-Parity RAID and Beyond, ACM Queue, December 2009

Any fixed redundancy level is eventually insufficient. We need to adapt.

Redundancy is Hard



Adam Leventhal, Triple-Parity RAID and Beyond, ACM Queue, December 2019

RAID rebuild now takes hours or days, and with 100+ disks, happens constantly. Must be online and low-overhead.

Data Integrity is Hard

- **Silent** data corruption is real.
  - Drive errors
  - OS/FS bugs
  - DMA errors
- Restoring from a backup takes too long.
  - 64TiB at 100MiB/s = about 8 days!
- Ensure integrity at runtime.
  - Signal corruption straight away and retry

Trends in Storage Integrity

- Uncorrectable bit error rates have stayed roughly constant
    - 1 in 10^{14} bits (~12TB) for desktop-class drives
    - 1 in 10^{15} bits (~120TB) for enterprise-class drives (allegedly)
    - Bad sector every 8-20TB in practice (desktop <u>and</u> enterprise)
- Drive capacities doubling every 12-18 months
- Number of drives per deployment increasing
- → Rapid increase in error rates
- Both silent and "noisy" data corruption becoming more common
- Cheap flash storage will only accelerate this trend

Measurements at CERN

- Wrote a simple application to write/verify 1GB file
  - Write 1MB, sleep 1 second, etc. until 1GB has been written
  - Read 1MB, verify, sleep 1 second, etc.
- Ran on 3000 rack servers with HW RAID card
- After 3 weeks, found <u>152</u> instances of <u>silent</u> data corruption
  - Previously thought “everything was fine”
- HW RAID only detected “noisy” data errors
- Need end-to-end verification to catch silent data corruption

Things got bigger!

ZFS

- The ‘Zettabyte’ File System
- Illustrates more modern design ideas:
  - Transactionality (Copy on Write)
  - Integrated volume management
  - Checksumming
- Developed at Sun (Solaris), before Oracle.
- Initially open-source (OpenSolaris).
- Oracle re-closed Solaris, and ZFS forked.

Copy on Write

- We want transactionality.
- Can we do better than journalling?
  - What atomic operations do we have?
  - How can we use them to build transactions?
- Parallels in Barrelfish and seL4
  - Maintain a tree invariant
  - Use the available atomicity

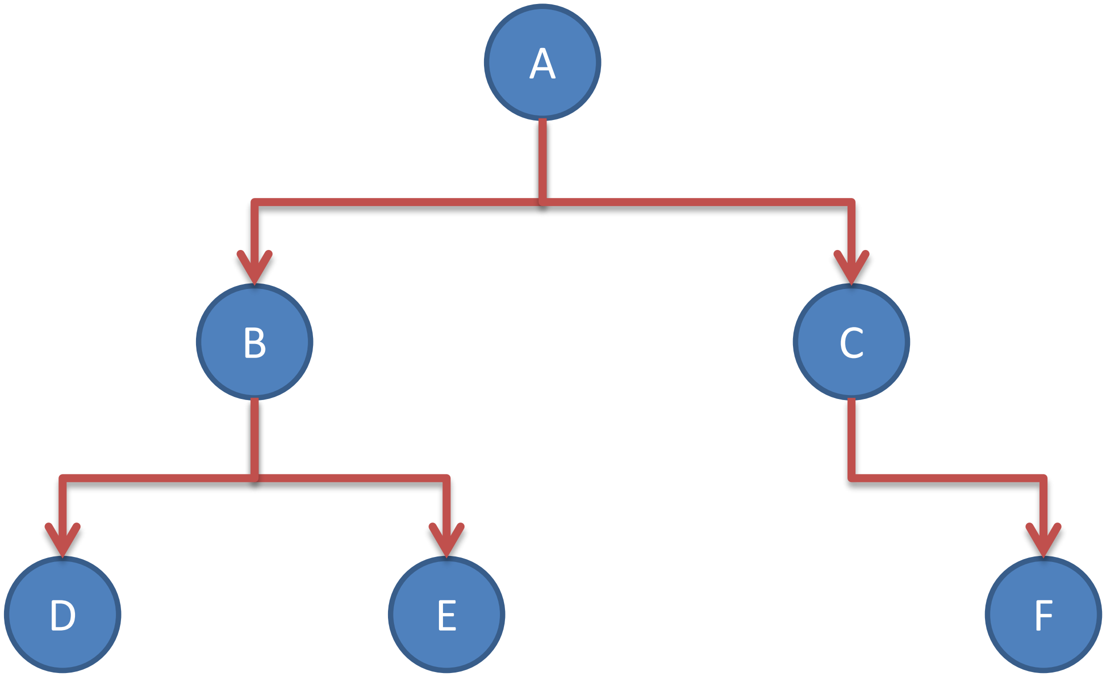
Preemption in seL4

- seL4 is a single-stack microkernel
  - No kernel threads
  - No continuations
- seL4 is a real-time system
  - The microkernel is preemptable
- seL4 has long-running operations
  - Recursive delete
  - Zeroing frames (to avoid information leaks)

Preemption in seL4

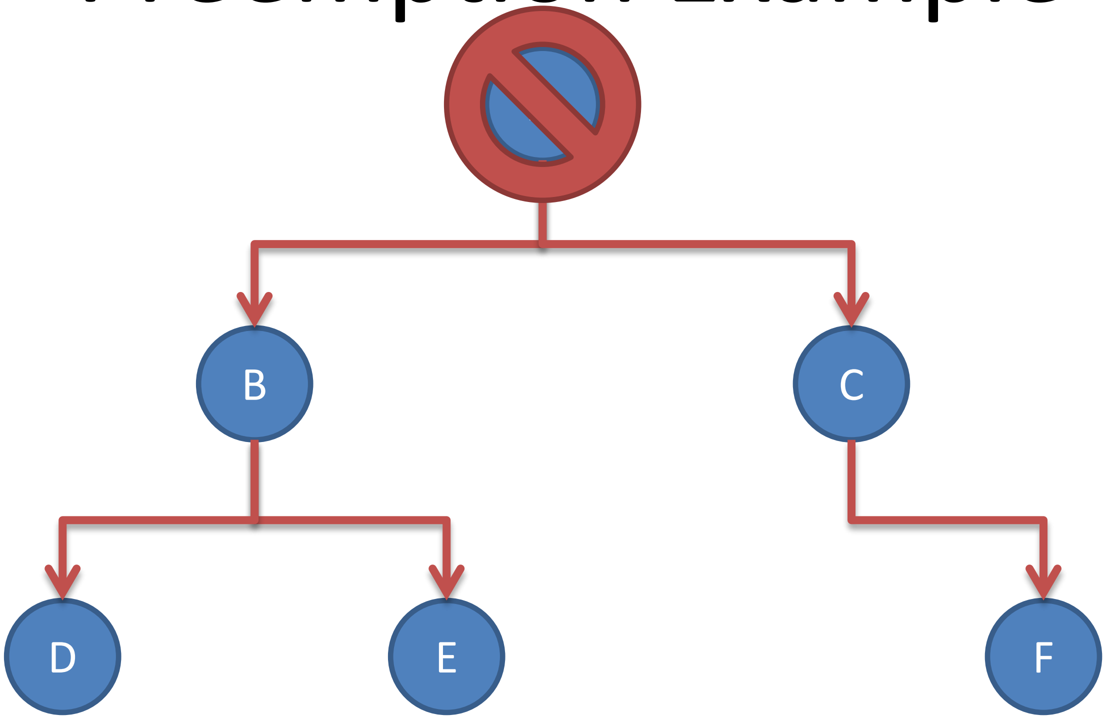
- Long operations have *preemption points*:
  - Where system invariants hold
  - Progress has been made
- Preempted operations *block* the calling thread
- If the thread is rescheduled, it continues
- If another thread touches the partially-deleted region, it restarts the blocked thread

Preemption Example



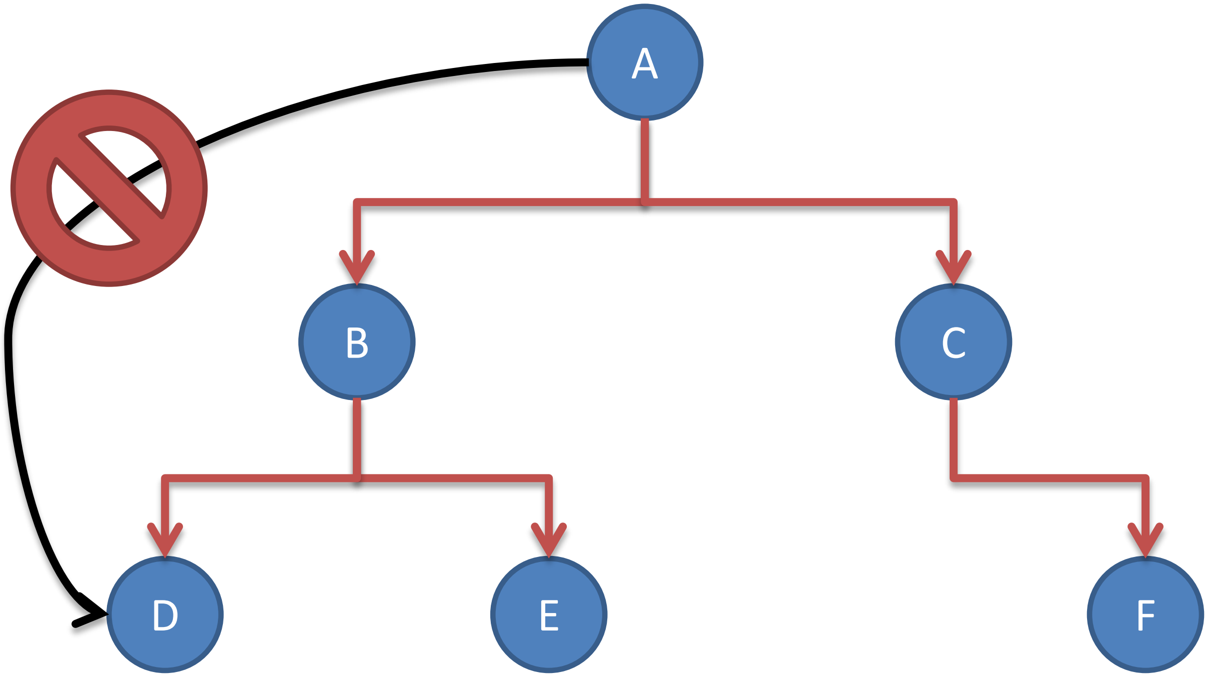
- Delete (revoke) the whole tree
  - Must delete children before parents
  - Tree may be arbitrarily deep

Preemption Example



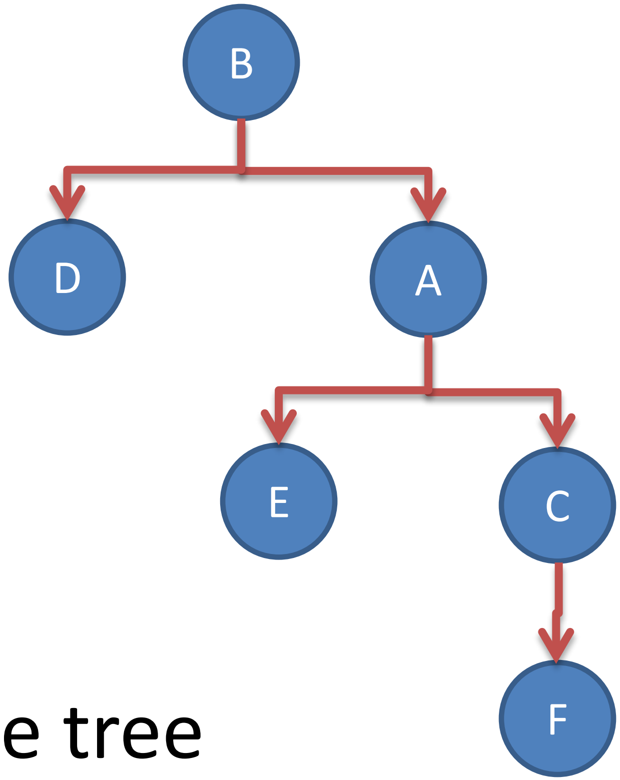
- Delete the whole tree
  - **Must delete children before parents**
  - Tree may be arbitrarily deep

Preemption Example



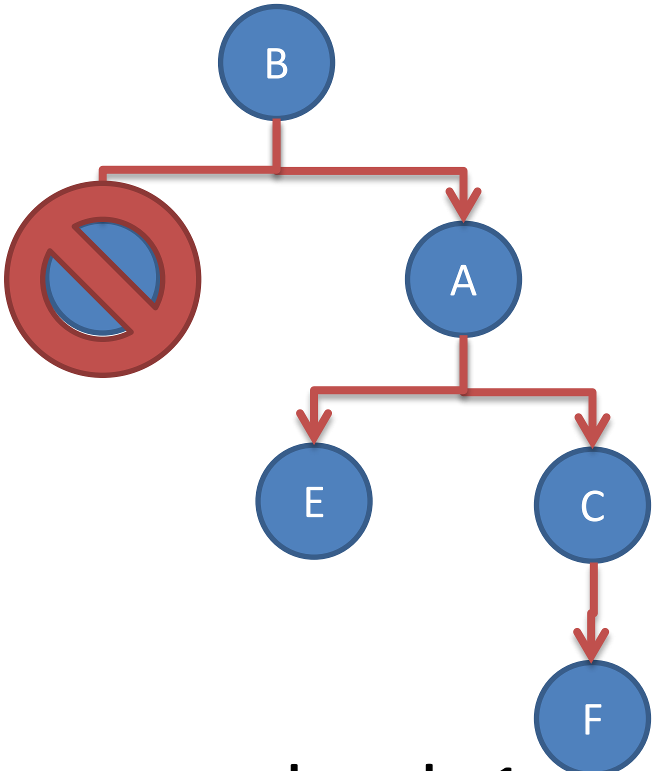
- Delete the whole tree
  - Must delete children before parents
  - **Tree may be arbitrarily deep**

Preemption Example



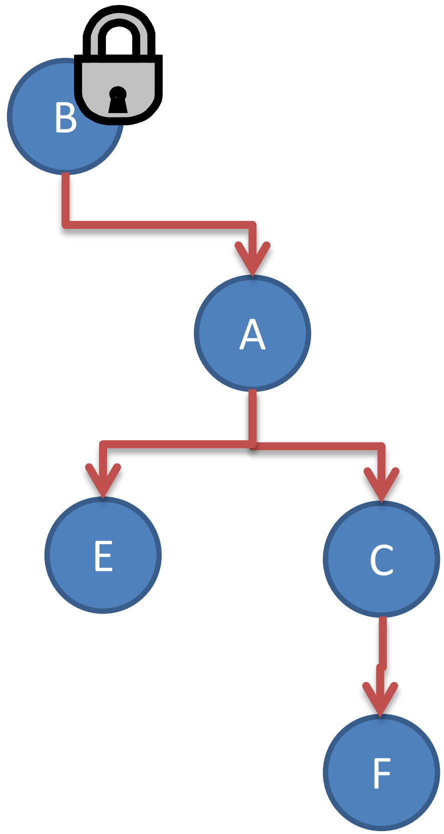
- Rotate the tree
  - Preserves the invariant: “No unrooted subtrees”
  - Only need to inspect up to depth 2

Preemption Example



- Delete leaves at depth 1
- It’s safe to preempt after any such step
  - What if another thread looks at B?

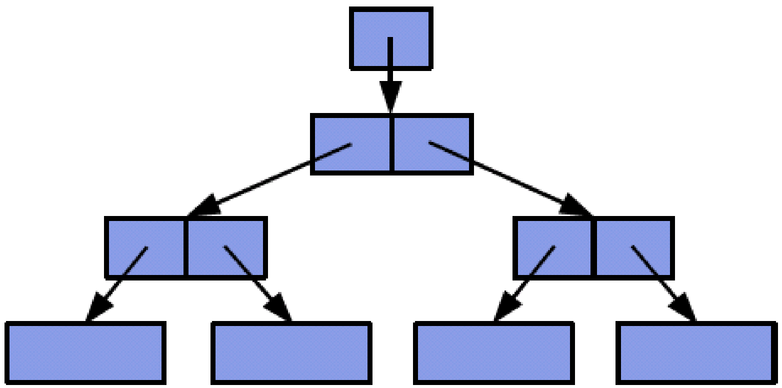
Preemption Example



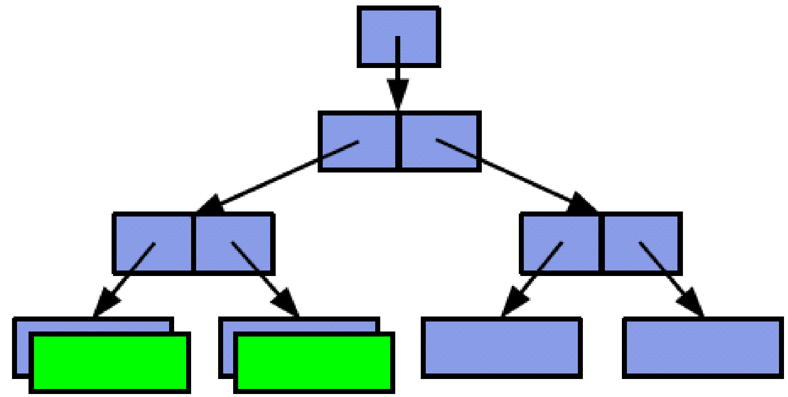
- Mark the root
  - If any thread enters the tree, it restarts the delete
  - No thread sees a partly-deleted tree

Copy-On-Write Transactions

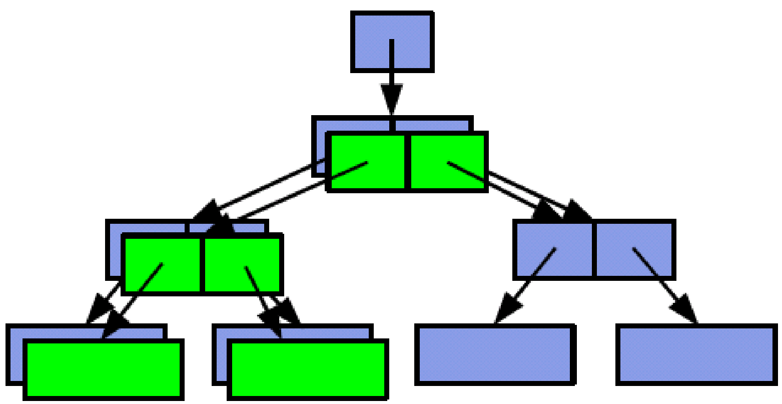
1. Initial block tree



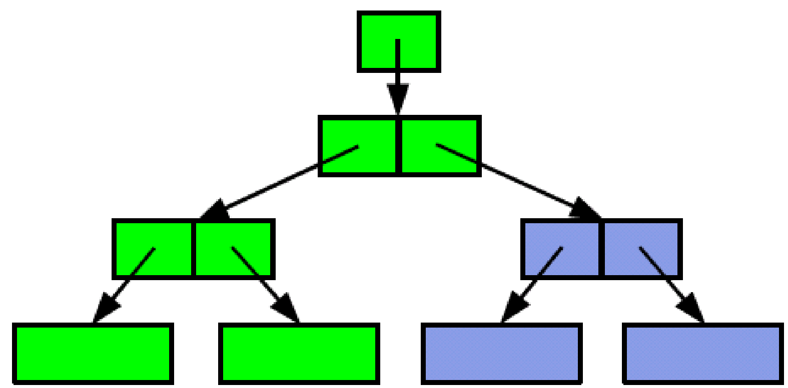
2. COW some blocks



3. COW indirect blocks

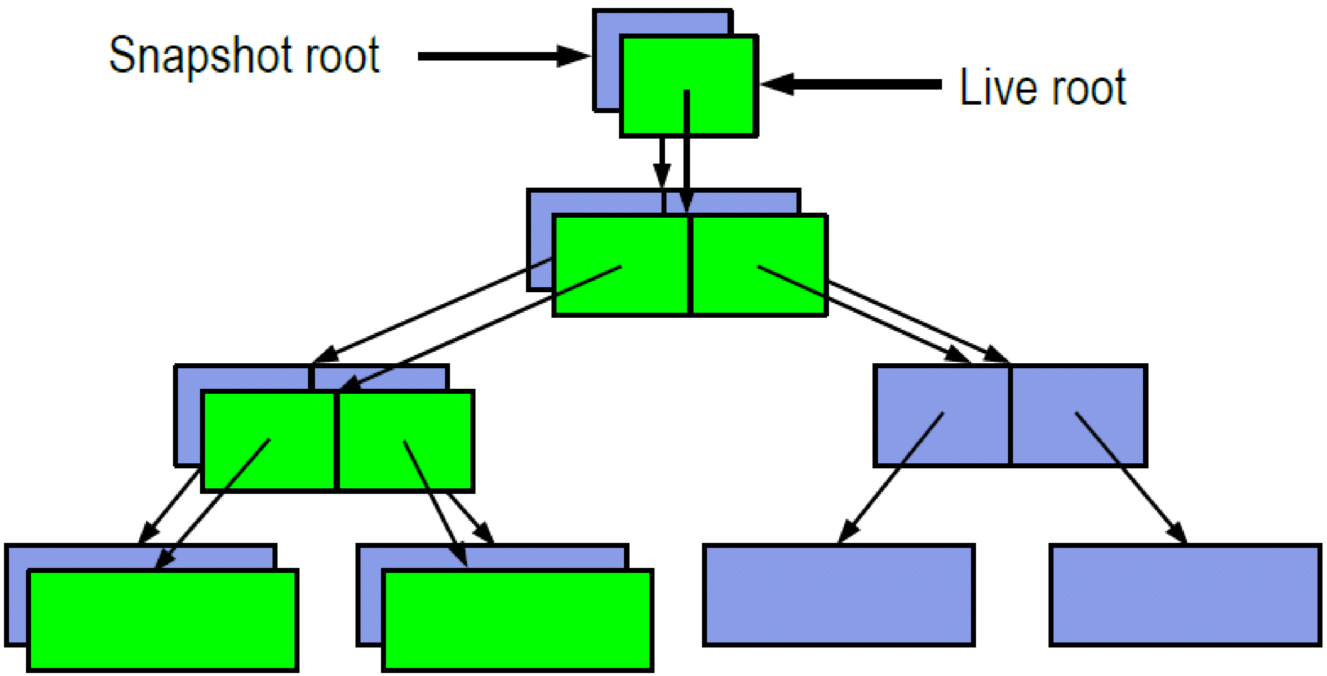


4. Rewrite uberblock (atomic)



Bonus: Constant-Time Snapshots

- At end of TX group, don't free COWed blocks
  - Actually cheaper to take a snapshot than not!



- The tricky part: how do you know when a block is free?

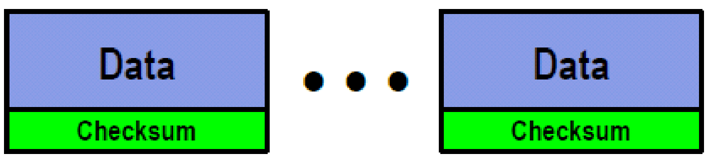
Checksumming

- RAID doesn't detect corruption very well:
  - Only if all copies are read (e.g. patrol scrub)
  - Can propagate errors during rebuild!
- Checksum data blocks:
  - Data becomes self-attesting.
  - Need to checksum metadata too.
  - Performance cost, but CPU is now pretty cheap.

End-to-End Data Integrity in ZFS

Disk Block Checksums

- Checksum stored with data block
- Any self-consistent block will pass
- Can't detect stray writes
- Inherent FS/volume interface limitation

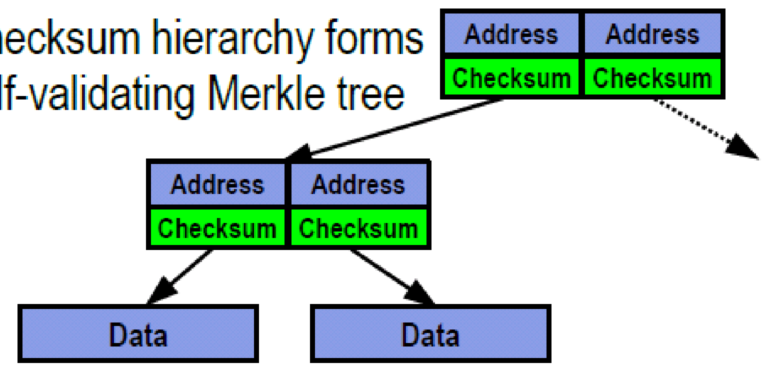


Disk checksum only validates media

- ✔ Bit rot
- ✗ Phantom writes
- ✗ Misdirected reads and writes
- ✗ DMA parity errors
- ✗ Driver bugs
- ✗ Accidental overwrite

ZFS Data Authentication

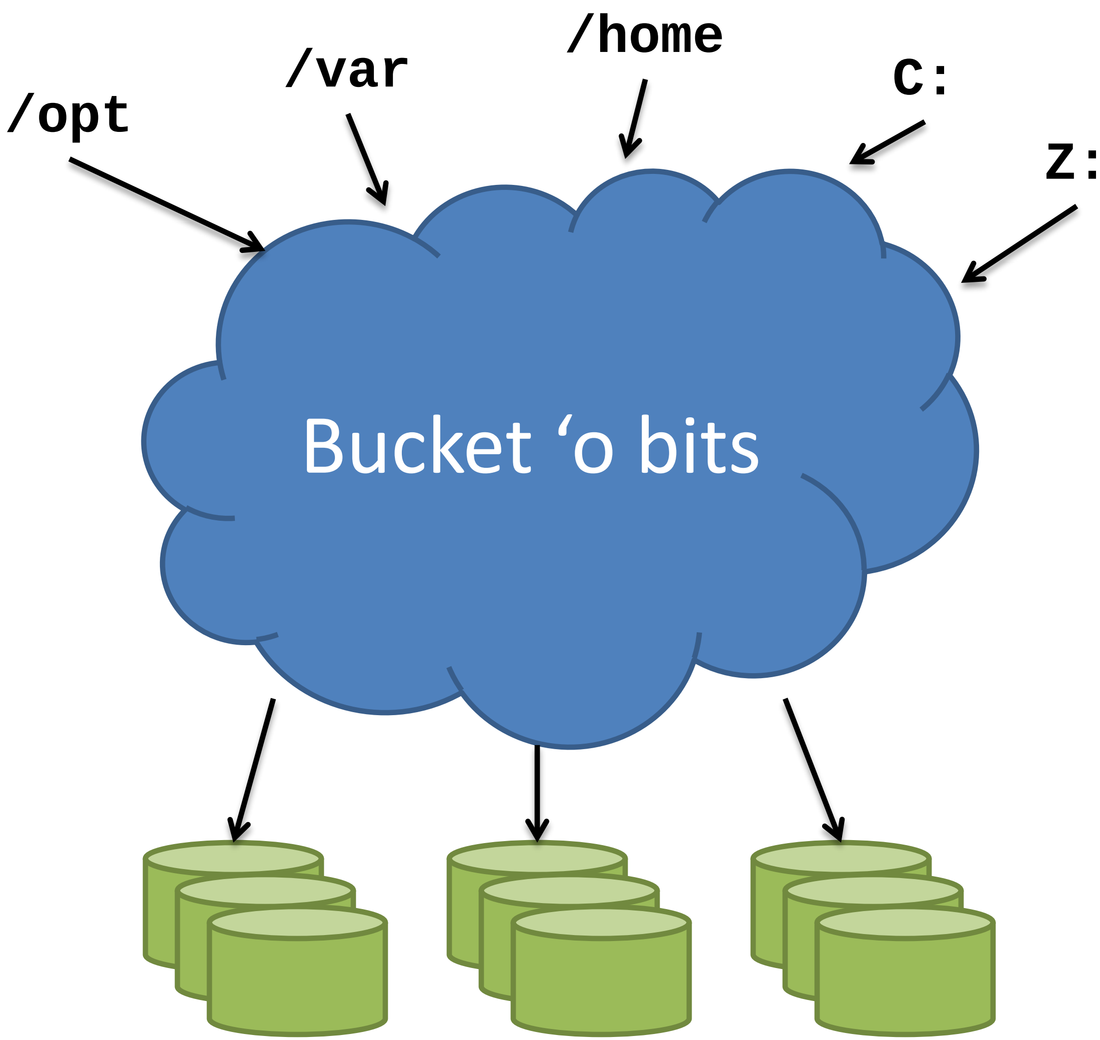
- Checksum stored in parent block pointer
- Fault isolation between data and checksum
- Checksum hierarchy forms self-validating Merkle tree



ZFS validates the entire I/O path

- ✔ Bit rot
- ✔ Phantom writes
- ✔ Misdirected reads and writes
- ✔ DMA parity errors
- ✔ Driver bugs
- ✔ Accidental overwrite

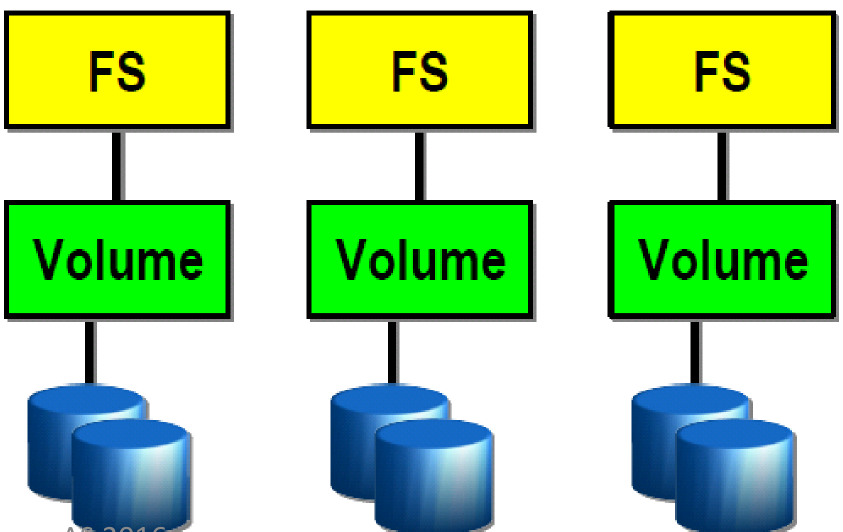
Storage Pools



FS/Volume Model vs. Pooled Storage

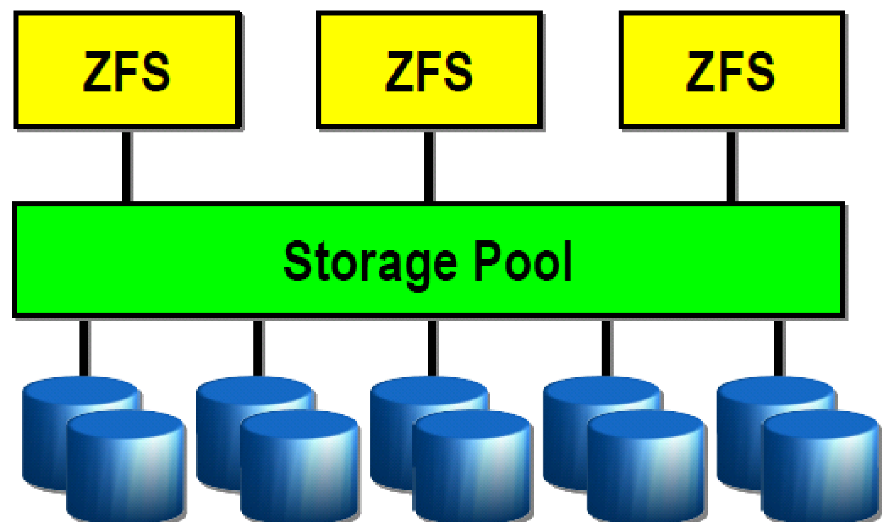
Traditional Volumes

- Abstraction: virtual disk
- Partition/volume for each FS
- Grow/shrink by hand
- Each FS has limited bandwidth
- Storage is fragmented, stranded



ZFS Pooled Storage

- Abstraction: malloc/free
- No partitions to manage
- Grow/shrink automatically
- All bandwidth always available
- All storage in the pool is shared



FS/Volume Interfaces vs. ZFS

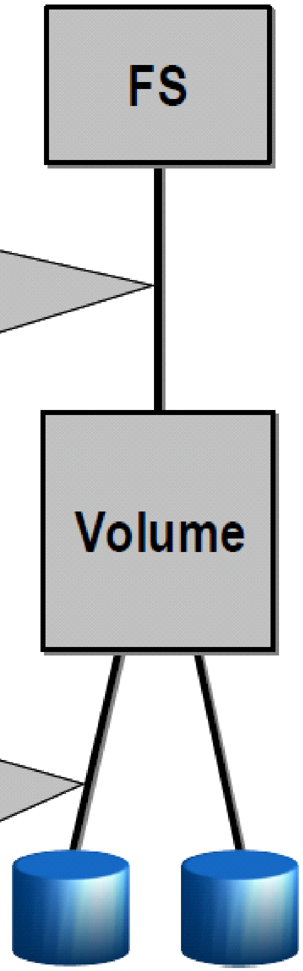
FS/Volume I/O Stack

Block Device Interface

- "Write this block, then that block, ..."
- Loss of power = loss of on-disk consistency
- Workaround: journaling, which is slow & complex

Block Device Interface

- Write each block to each disk immediately to keep mirrors in sync
- Loss of power = resync
- Synchronous and slow



ZFS I/O Stack

Object-Based Transactions

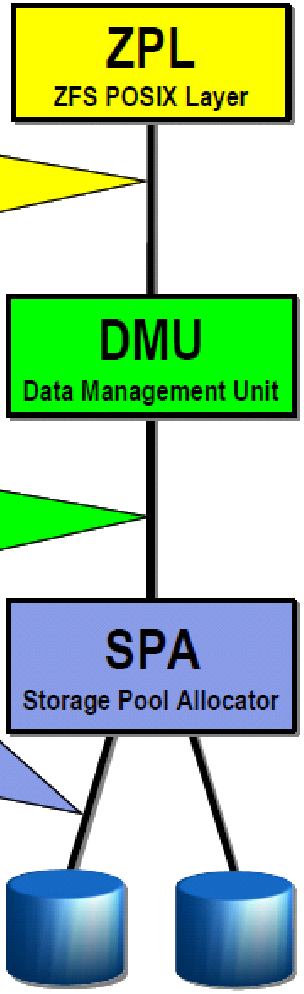
- "Make these 7 changes to these 3 objects"
- Atomic (all-or-nothing)

Transaction Group Commit

- Atomic for entire group
- Always consistent on disk
- No journal – not needed

Transaction Group Batch I/O

- Schedule, aggregate, and issue I/O at will
- No resync if power lost
- Runs at platter speed

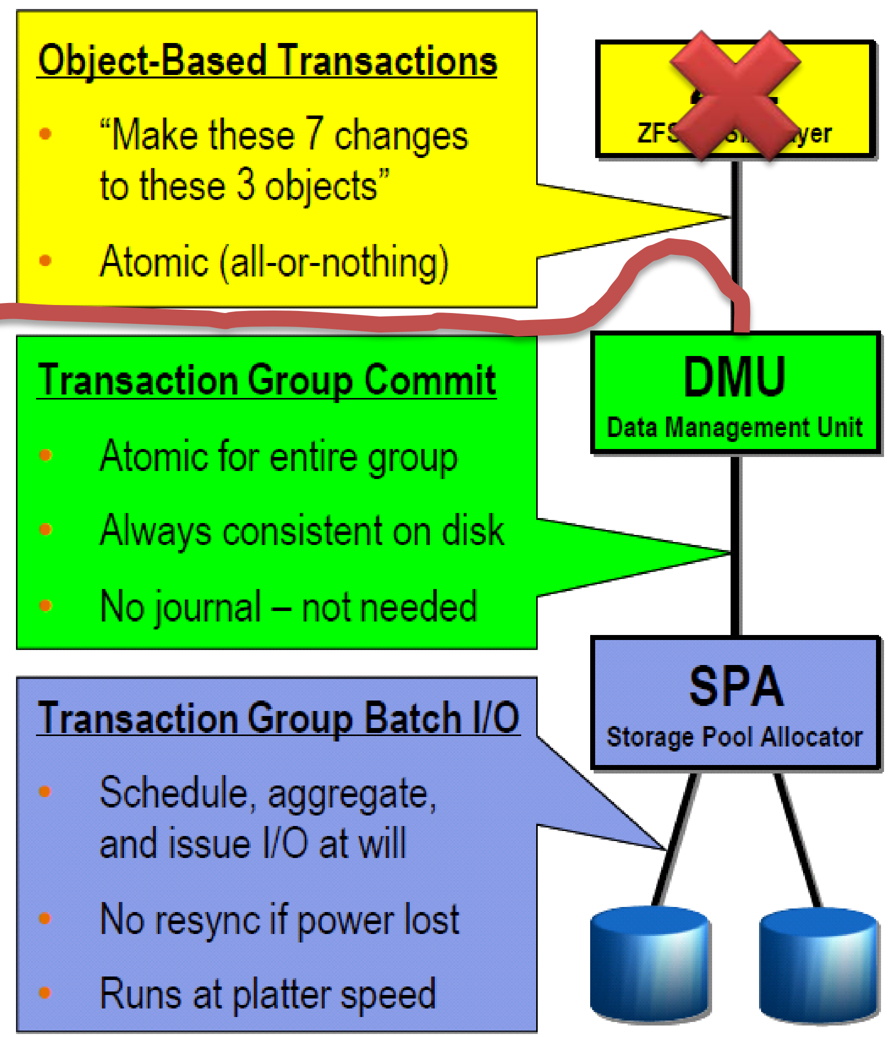


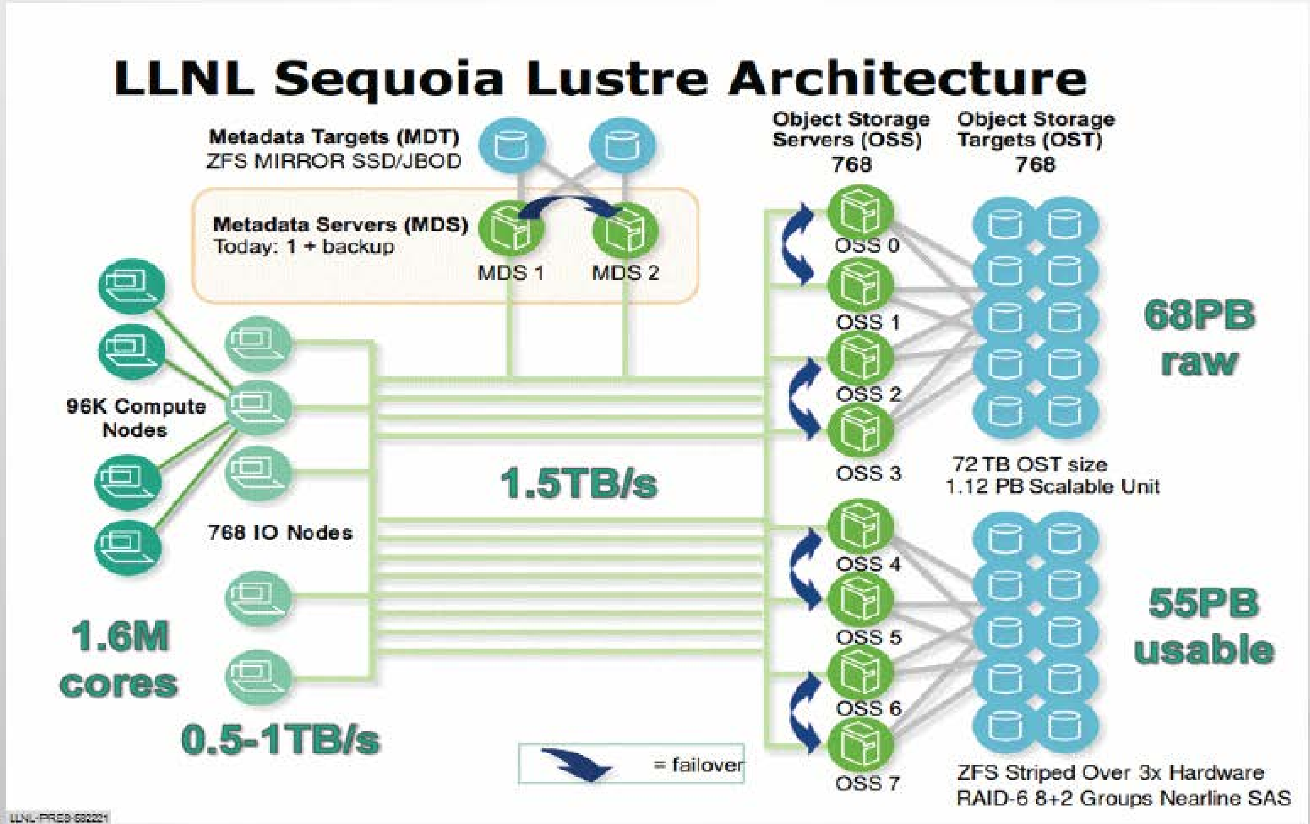
ZFS Object Layer as a Backend

Lustre

LLNL replaced the ZPL with a backend for the Lustre filesystem for their supercomputers.

ZFS I/O Stack



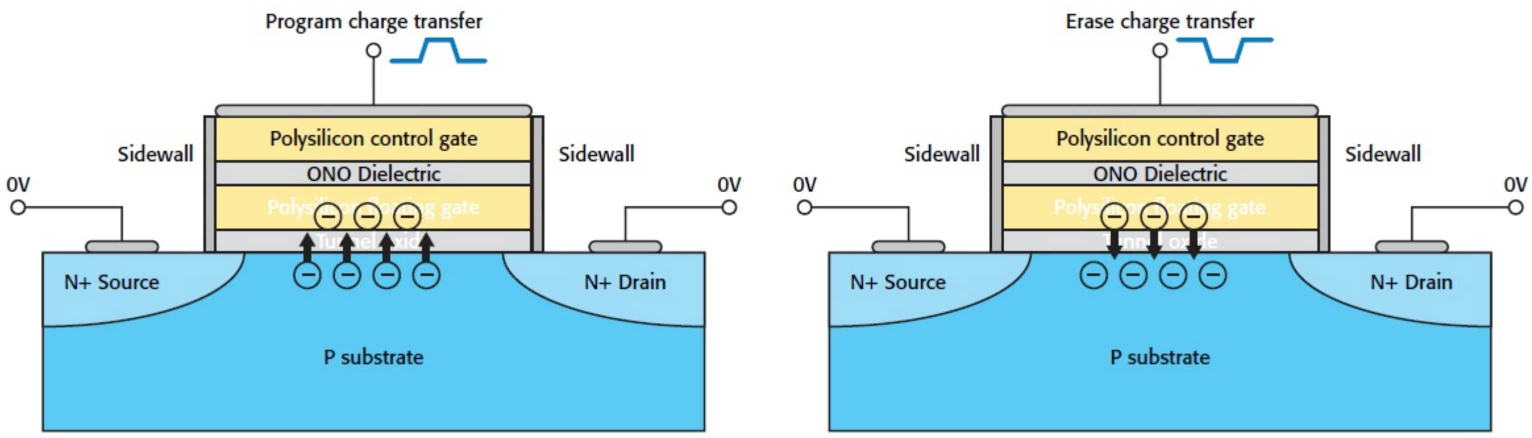


55 petabyte storage
850 gigabytes/sec measured sustained write throughput
768 OSSs & OSTs
Each OST is 72 terabytes

ZFS Features and benefits

- Copy on Write (COW) serializes random writes
  - Performance no longer bounded by drive IOPS
- Single volume size limit of 16 exabytes
- Zero fsck time. On-line data integrity and error handling
- Expensive RAID controllers are unnecessary
- Data is always checksummed and self repairing to avoid silent corruption.
- Easy aggregation of multiple devices in to a single OST.
- A 256 zettabytes (2^78 bytes) OST size limit enables larger servers.
- Snapshot the Lustre file system prior to maintenance for worry free updates.
- Transparent compression increases your total usable capacity.

Flash Memory

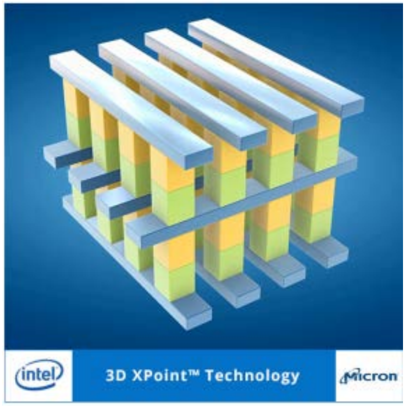


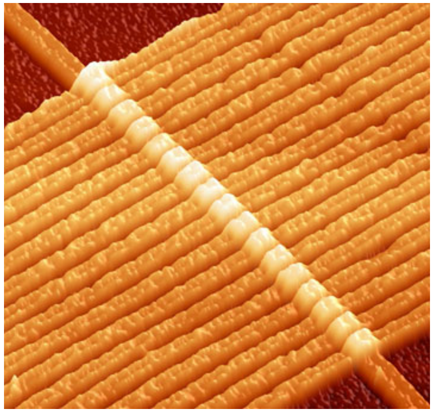
- Moderately higher bandwidth than HDDs, 300-1000MiB/s
- Very low random read latency: 100us, high write latency is amortised
- Limited write endurance
- Roughly 10x more expensive

Effects on FS Design

- We can just stick ext4 on an SSD and call it a day, but that's just dumb:
  - The performance tradeoffs are totally different: we don't need to avoid seeks any more.
  - …but we do need to avoid writes!
- Should an SSD pretend to be a disk at all?
  - Should it be a transparent cache?
  - Should it just be addressed like memory?

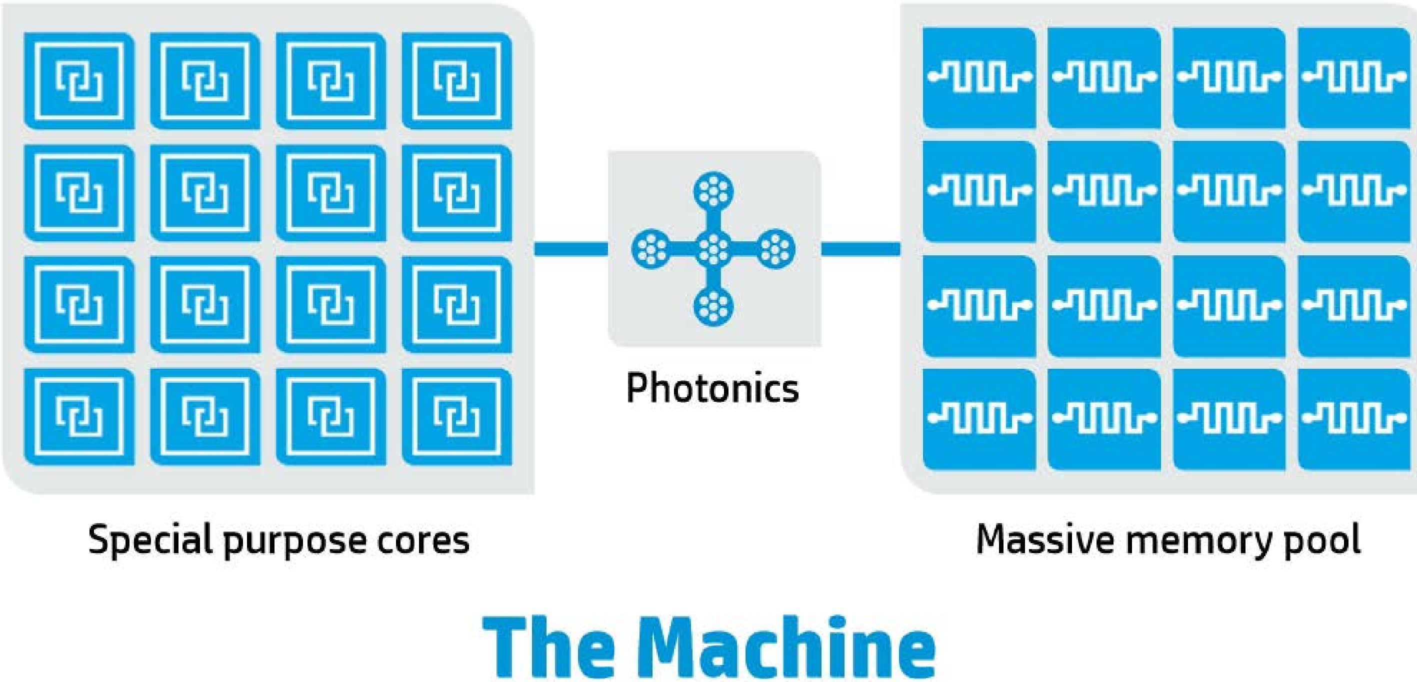
Fast NVM





- It’s coming:
  - Memristors, X-point (PCM, we think).
  - Almost as fast as DRAM, persistent, low power.
  - Do we even need a filesystem anymore?

The Machine



- Lots and lots of NVM, not colocated with CPUs