

**ITEO**

# Moderne File Systeme

Bruno Joho  
[bruno.joho@hslu.ch](mailto:bruno.joho@hslu.ch)

# Lernziele

- Sie wissen was ein modernes Filesystem leisten muss.
- Sie wissen wie ein modernes Filesystem aufgebaut ist.
- Sie wissen wie der Kern und das File System zusammenarbeiten.
- Sie kennen die internen Strukturen eines Filesystems.

# Inhalt



- Buffer Cache
- Virtual File System (VFS)
- moderne Filesysteme
- ZFS Struktur
- Copy on Write
- Snapshots
- RAID
- Skalierbarkeit

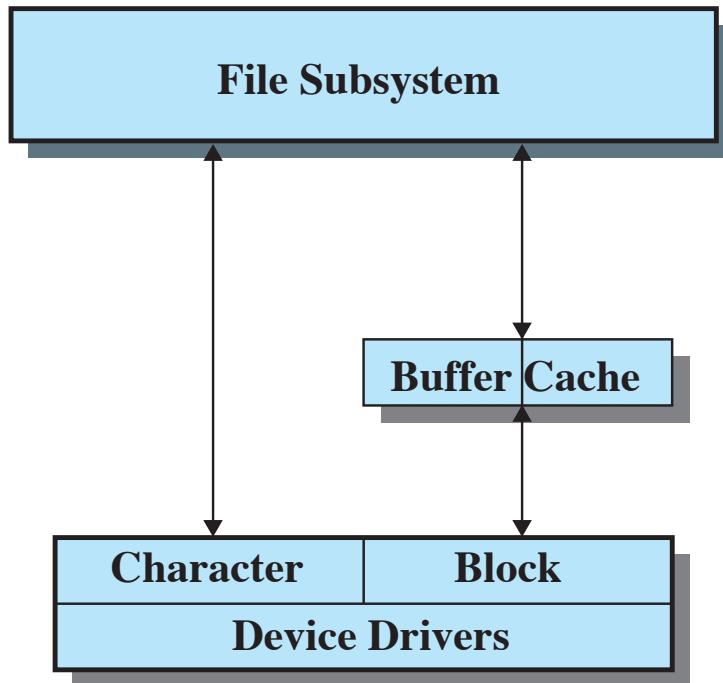
# Buffer Cache

- Wenn ein Prozess auf Datenblöcke eines Files zugreifen will, so bringt der Kern diese Blöcke in den Hauptspeicher, ändert diese nach Bedarf und macht danach eine Anfrage dieser im File System zu speichern.
- Beispiel: copy cp fileone.c filetwo.c
- Um den Durchsatz zu erhöhen und die Antwortzeiten zu verkleinern, verringert der Kern die Disk Zugriffszeiten indem er einen Pool von internen Daten Puffer unterhält. Sogenannte **Buffer Cache**.

# Buffer Cache im Kern

Erinnern Sie sich? Vorlesung *System SW*.  
Nebenstehend ein Block aus dem Kern  
Architektur Plan.

- Das File Subsystem „cached“ Blöcke vom Filesystem.
- Blöcke werden in einen Buffer Cache (Struktur im Memory) abgelegt und eine Zeitlang verwaltet.
- Buffer Cache beschleunigen den Blockzugriff, wenn er zuvor schon einmal verwendet wurde.



# Was ist ein Buffer Cache

- Ist eigentlich ein Disk Cache.
- I/O Operationen werden durch den Buffer Cache abgehandelt.
- Der Datentransfer zwischen dem Buffer Cache (Kern-Raum) und dem User Prozess-Raum wird immer über DMA\* gemacht.
  - Braucht keine Prozessor Zyklen
  - Braucht aber Bus Zyklen
- **3 verkettete Listen** werden gebraucht um den Buffer Cache zu unterhalten:
  - **Freie Liste**: listet alle Plätze vom Cache die für den Gebrauch zur Verfügung stehen.
  - **Device Liste**: listet alle Buffer die momentan mit einem FS verknüpft sind.
  - **Driver I/O Queue**: listet die Buffer auf die bei einem bestimmten Device einen I/O ausführen oder auf einen I/O warten.

\* DMA = direct memory access. Modul IS, Kap. 4.1 BS, „Interaktion mit I/O Geräten“

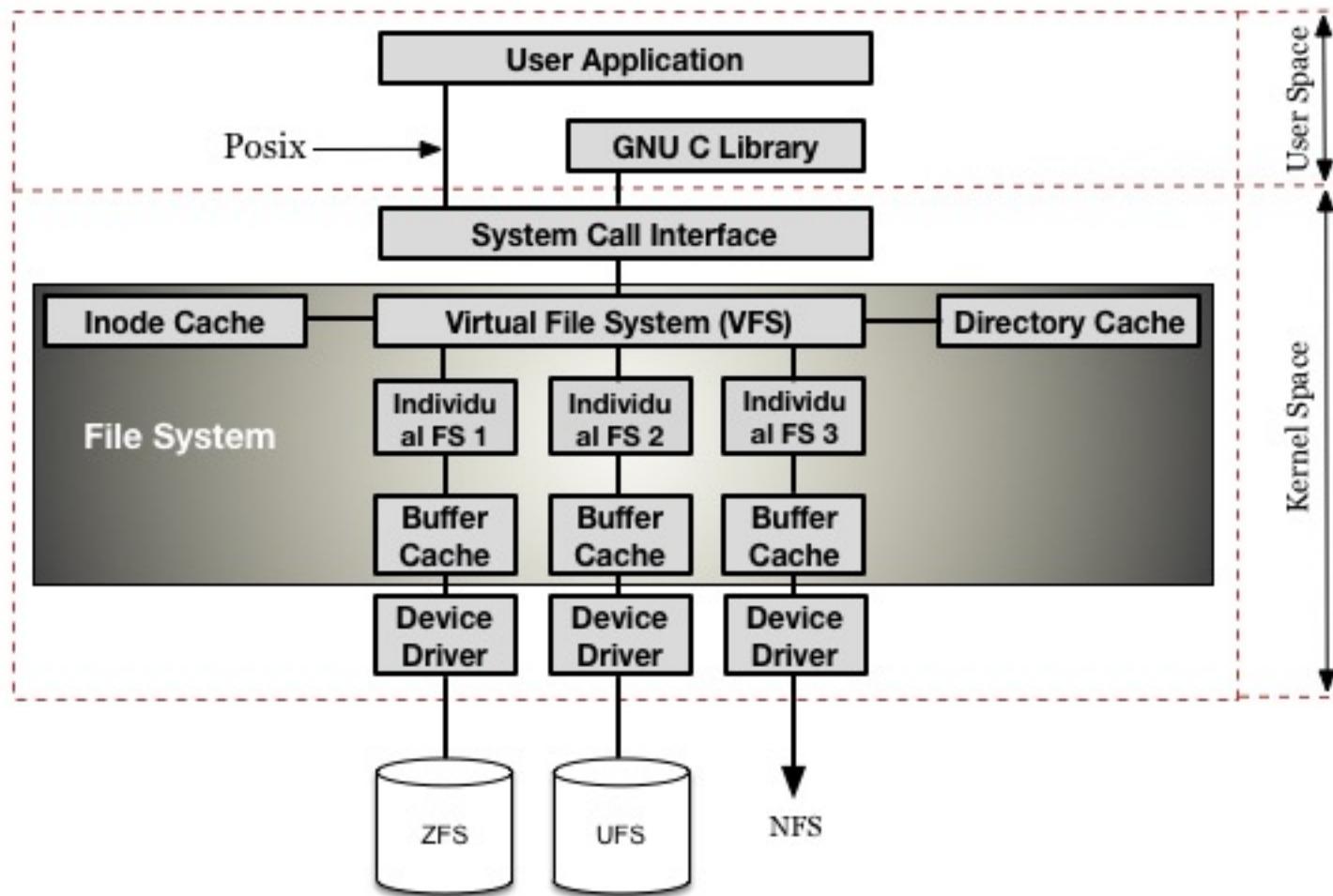
# Funktionsweise vom Buffer Cache

- Buffer Cache enthalten die Daten von kürzlich verwendeten Disk Blöcken.
- Wenn der Kern Daten von der Disk lesen will versucht er sie zuerst vom Buffer Cache zu lesen.
- Falls die Daten im Buffer Cache sind braucht er diese nicht von der Disk zu lesen.
- Falls die Daten nicht im Buffer Cache sind so liest der Kern diese von der Disk und legt sie im Buffer Cache ab.

# Inhalt

- Buffer Cache
- Virtual File System (VFS)
- moderne Filesysteme
- ZFS Struktur
- Copy on Write
- Snapshots
- RAID
- Skalierbarkeit

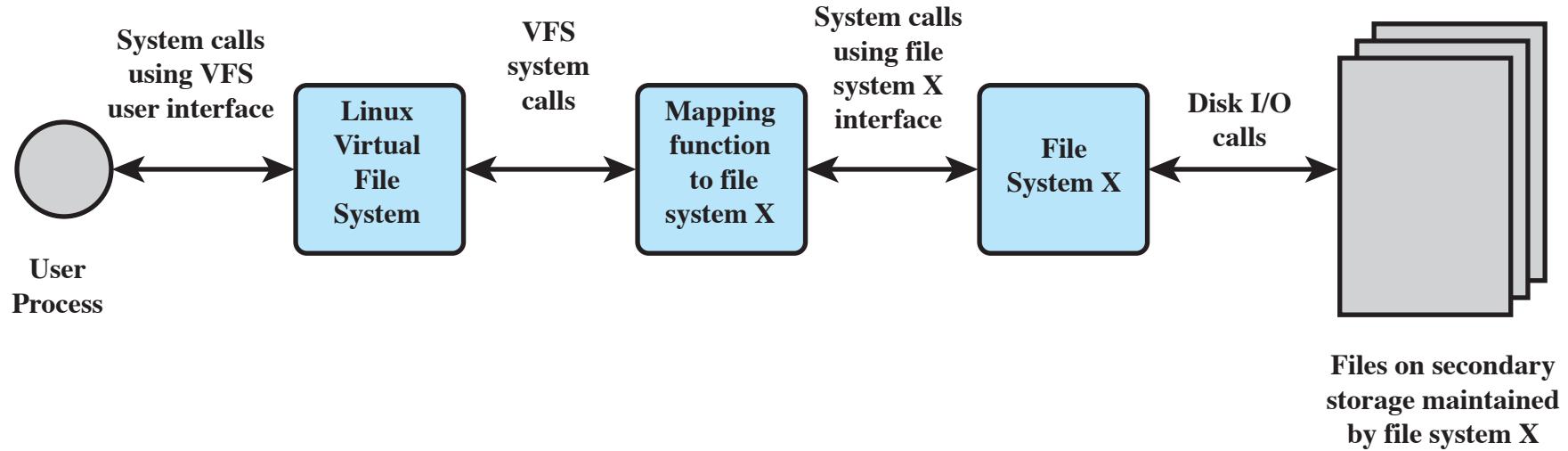
# Virtual File System



# Unix Virtual File System

- VFS präsentiert ein einziges uniformes FS Interface zum User Prozess
- Der User Prozess setzt einen FS call im POSIX Format ab.
- VFS setzt voraus:
  - dass ein File einen symbolischen, einzigartigen Namen hat und von einem Directory identifiziert werden kann.
  - Ein File hat einen Besitzer und ist geschützt gegen nicht autorisierten Zugriff.
- Für jedes spezifische FS wird ein mapping Modul bereitgestellt wo das reelle FS auf das virtuelle FS transformiert wird.

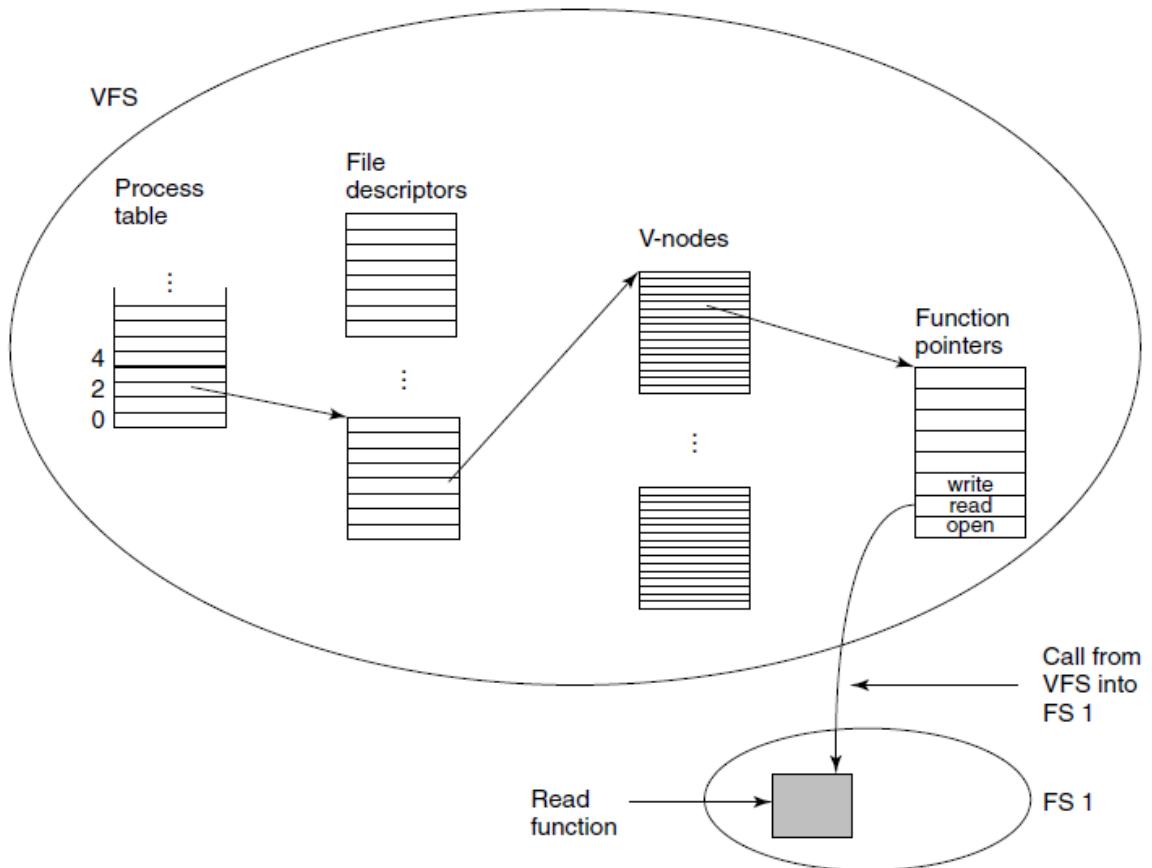
# VFS Konzept



- User FS call wird in einen FS spezifischen Aufruf umgeformt.
- Das target FS registriert sich beim VSF (stellt Funktionen zur Verfügung).
- Die mapping Funktion muss in der Lage sein dynamisch die Files korrespondierend zum Directory zu rekonstruieren.
- Beispiel: `open ( „/usr/include/unistd.h“, O_RDONLY)`

# step by step

1. VFS sucht nach entsprechendem Superblock der registrierten FS (mount)
2. VFS findet root Verzeichnis und findet über Pfad die iNnode von Datei.
3. VFS kopiert iNode in vNode (RAM) Struktur.
4. vNode [2] Struktur enthält Pointer auf Funktionstabelle (FS spezifisch).
5. VFS erstellt für den aufrufenden Prozess einen Eintrag im File Deskriptor
6. User Prozess greift über File Deskriptor, vNode, und Funktions-Pointer auf das File zu.



Vereinfachte Sicht auf die Datenstrukturen wie sie im VFS und konkreten FS verwendet werden um ein read auszuführen.

# Inhalt

- Buffer Cache
- Virtual File System (VFS)
- moderne Filesysteme
- ZFS Struktur
- Copy on Write
- Snapshots
- RAID
- Skalierbarkeit

# **moderne Filesysteme**

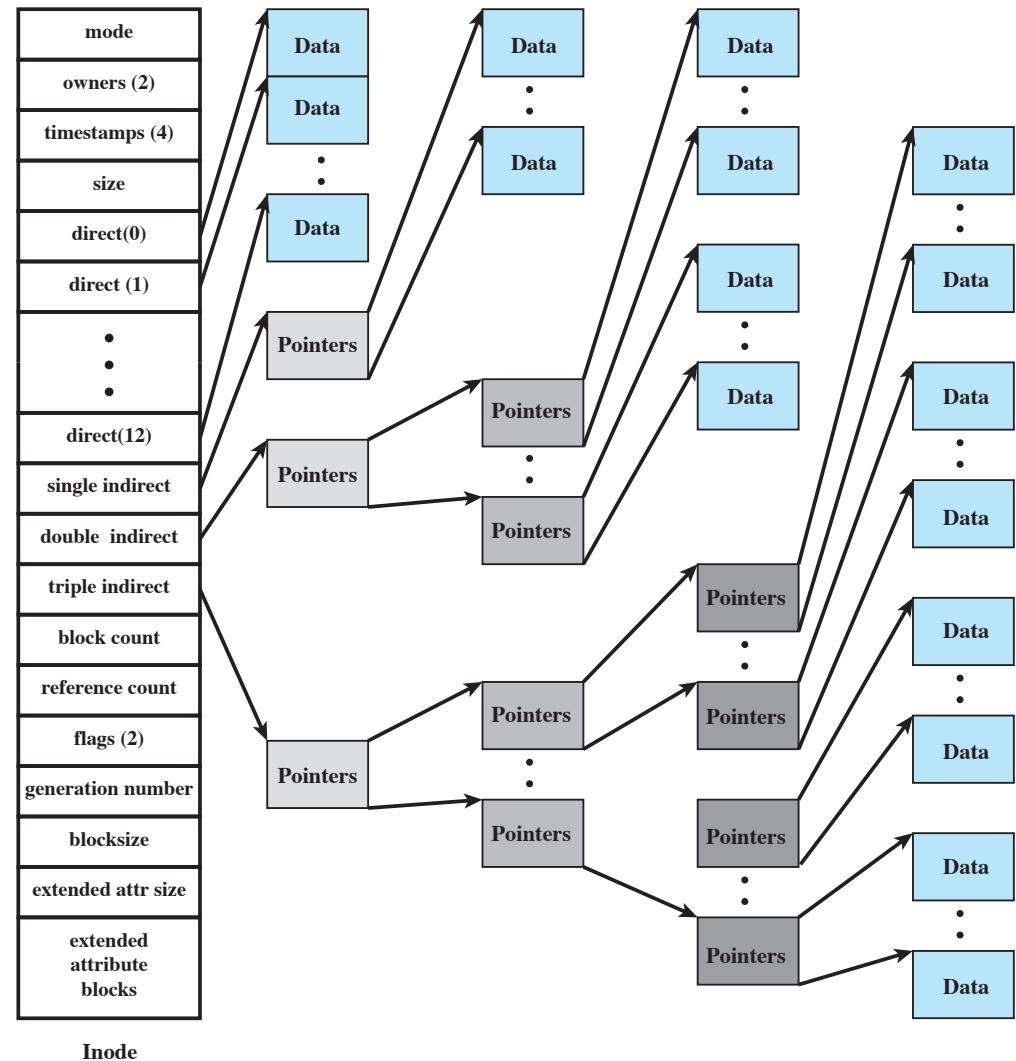
Wir betrachten im folgenden Sun/Oracle's zfs File System als Vertreter eines modernen Filesystems.

## **Anforderungen:**

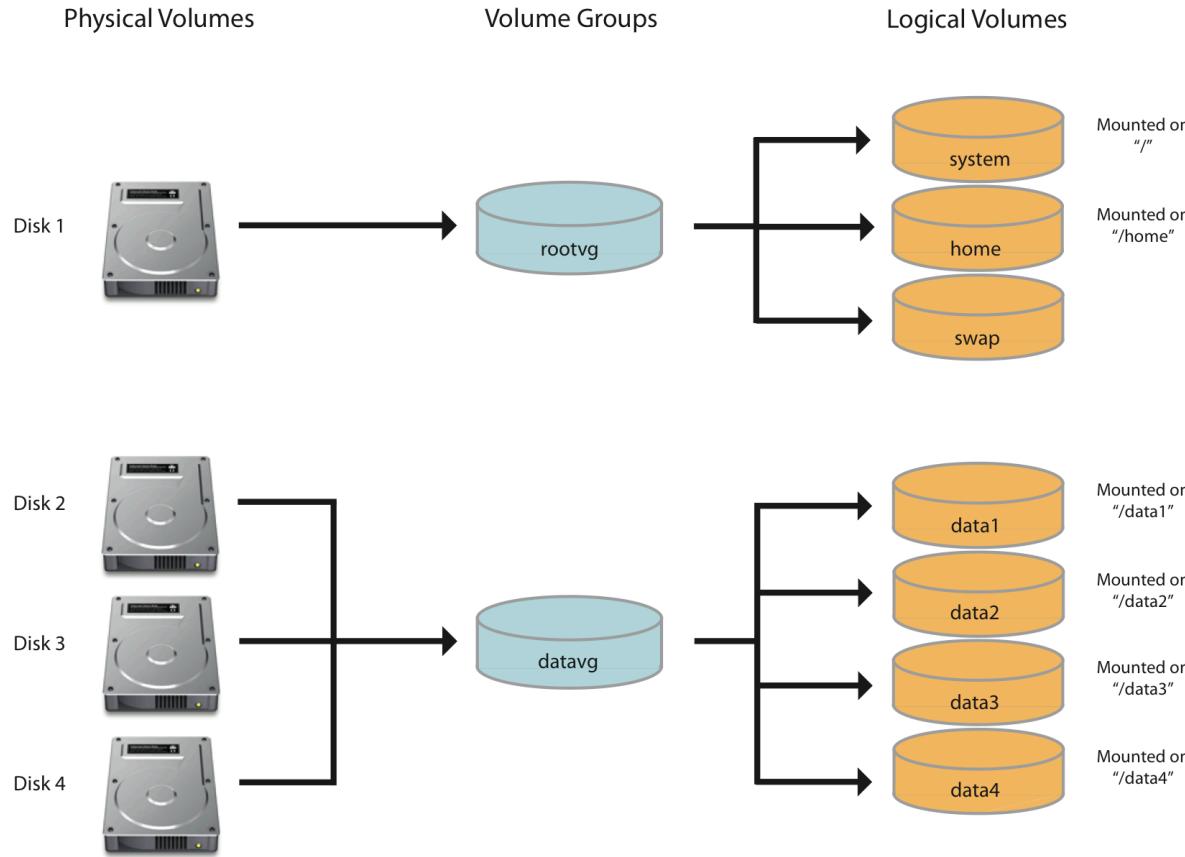
- grosse Adressierbarkeit
- Volume Manager eingebaut
- umfangreiche Funktionalitäten
- immer konsistent und integer
- kein fsck mehr nötig nach Absturz
- kann mit „silent corruption“ umgehen

# Zuerst war UFS

Erinnern sie sich?: Computer  
& Network Architecture  
Kapitel 4.3: „UFS“

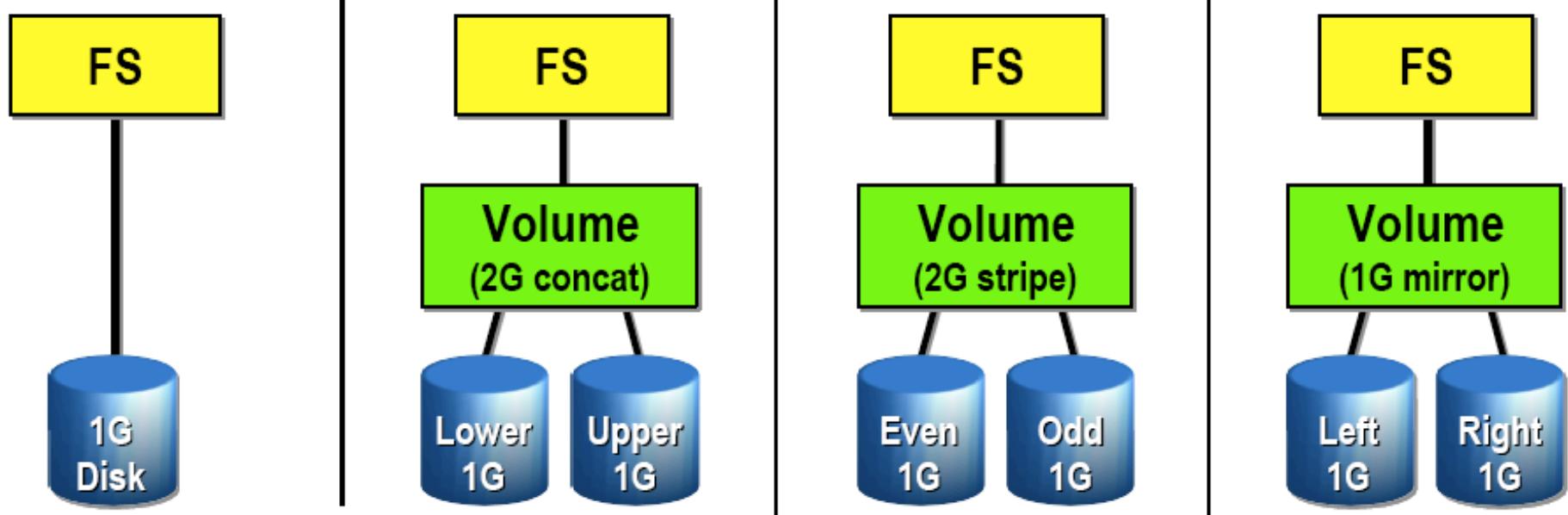


# Volumes

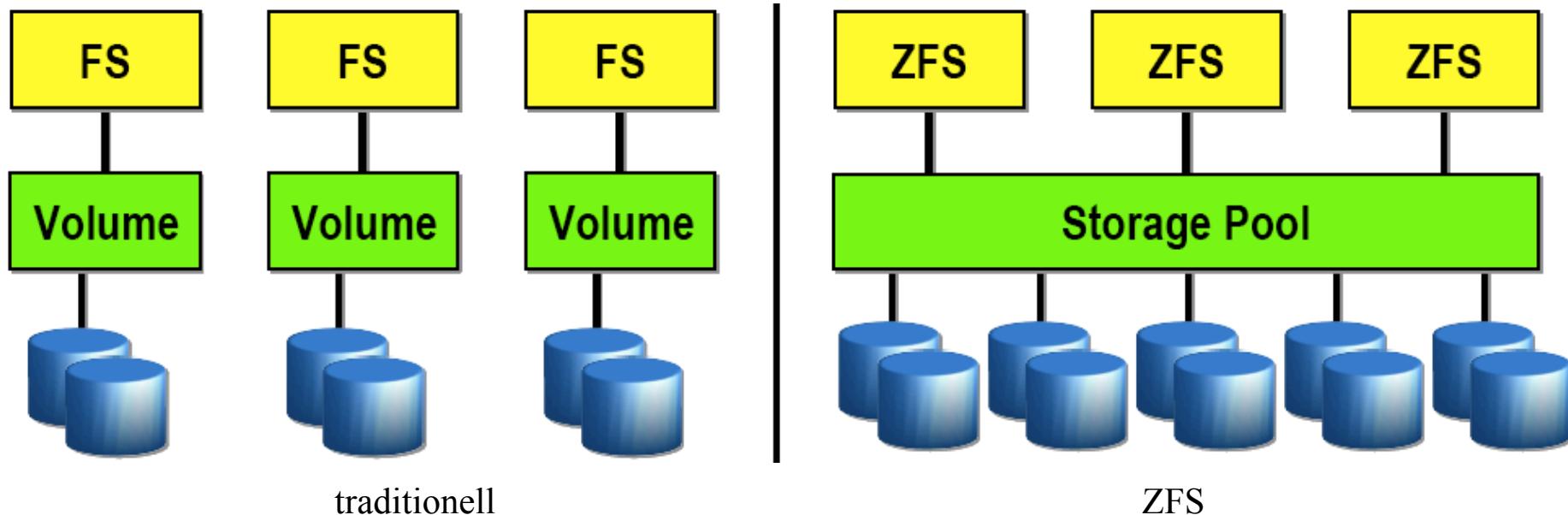


Volumes fassen verteilte Speicherblöcke zu logischen Einheiten, sog. Volumes zusammen.

# Volumes (2)



# Volumes vs. Pooled Storage

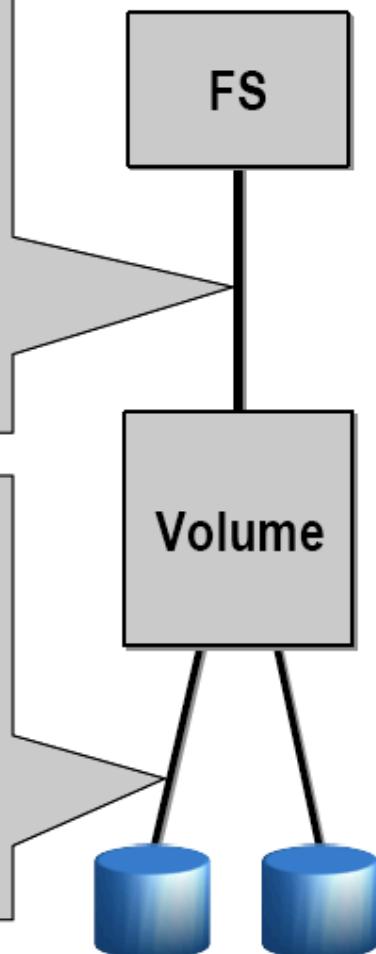


# FS Volume Interface vs. ZFS

## 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)

ZPL

ZFS POSIX Layer

DMU

Data Management Unit

SPA

Storage Pool Allocator

### Transaction Group Commit

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

### Transaction Group Batch I/O

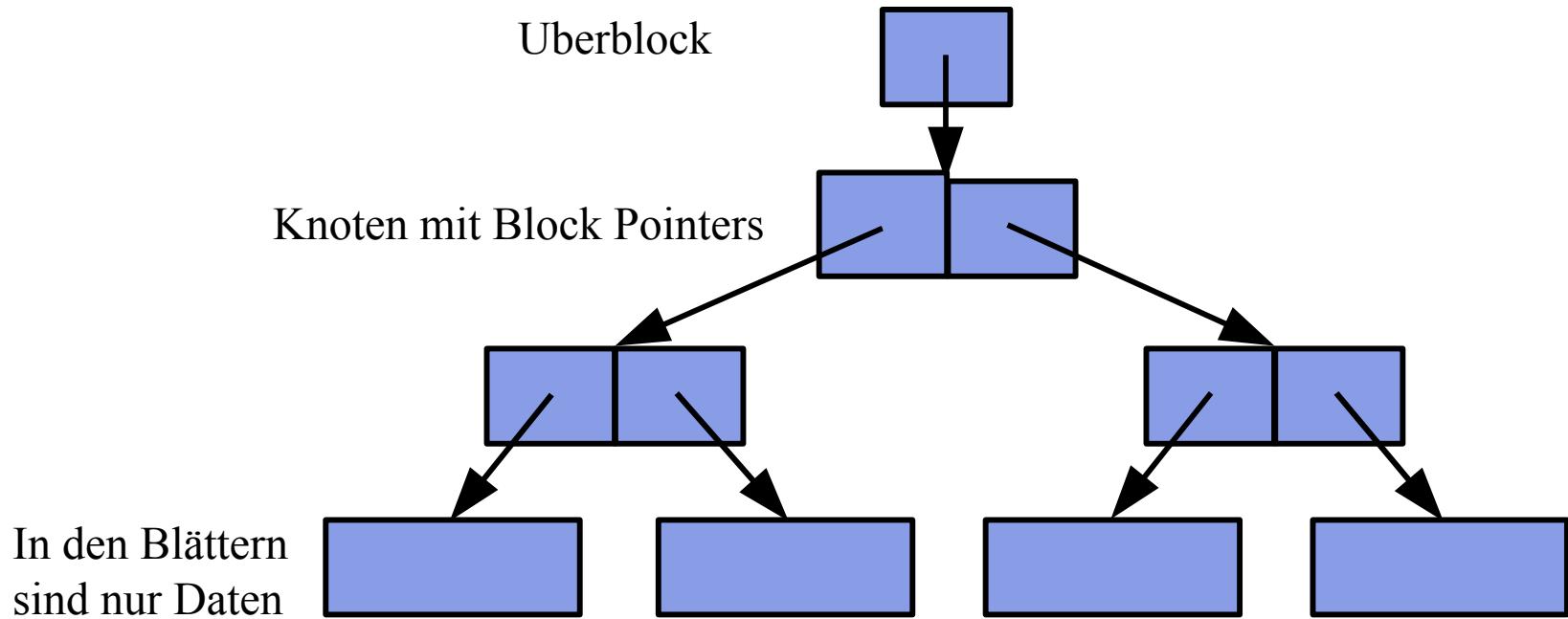
- Schedule, aggregate, and issue I/O at will
- No resync if power lost
- Runs at platter speed

# Inhalt

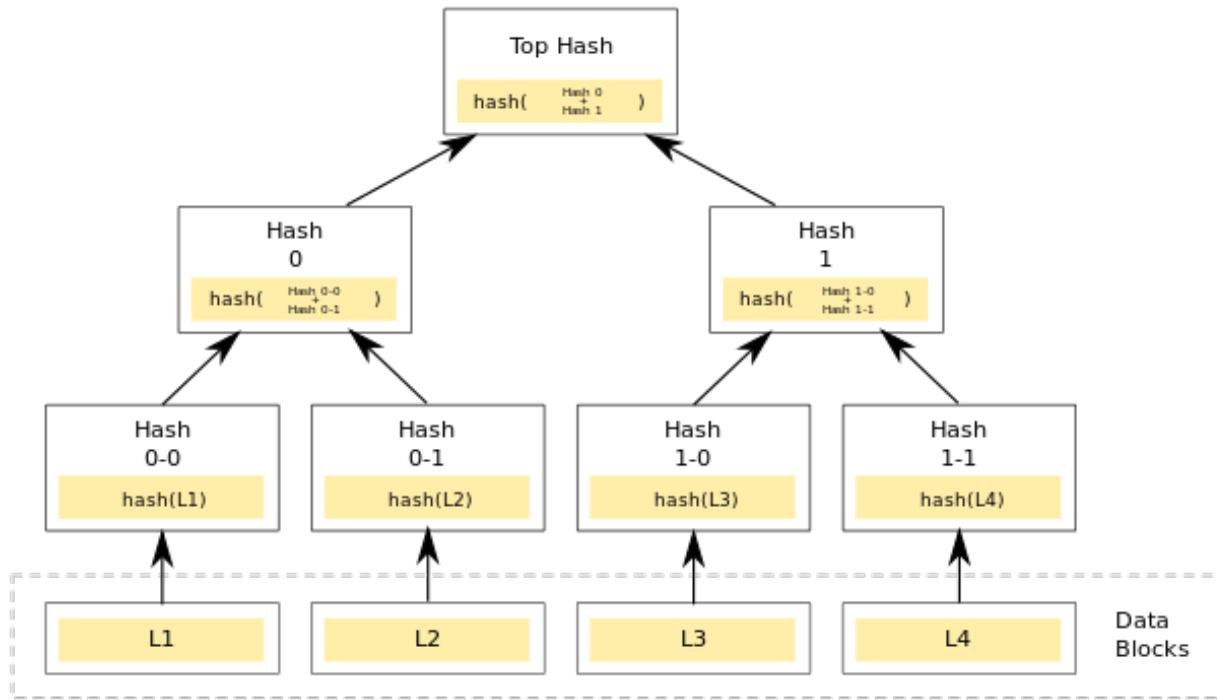
- Buffer Cache
- Virtual File System (VFS)
- moderne Filesysteme
- ZFS Struktur
- Copy on Write
- Snapshots
- RAID
- Skalierbarkeit



# ZFS Struktur

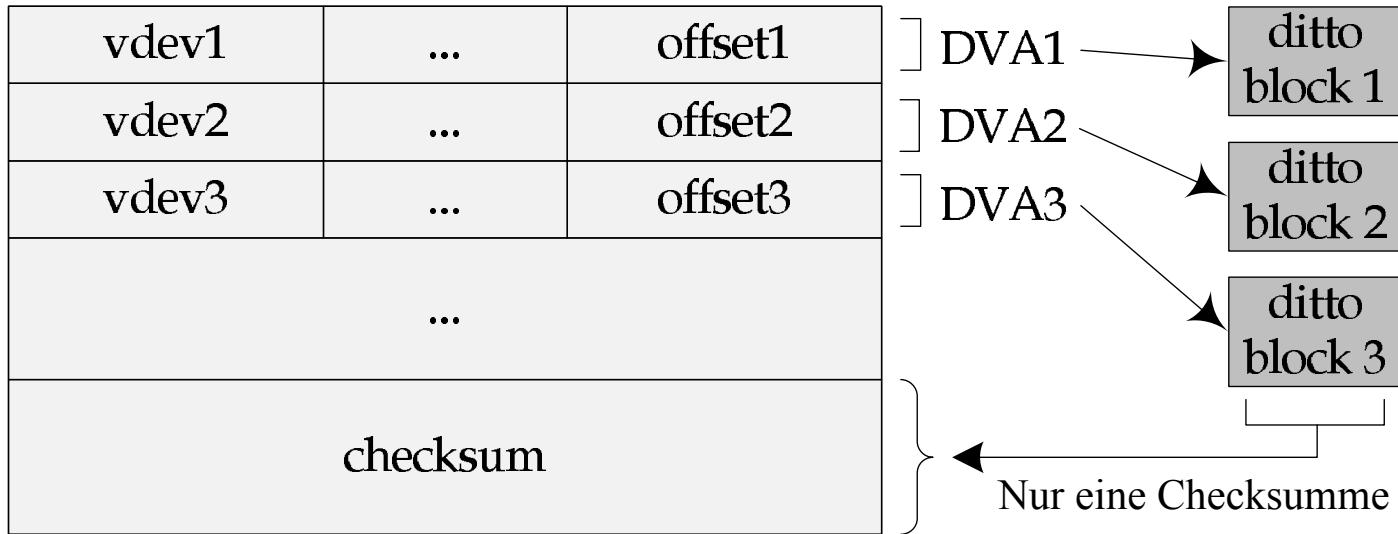


# ZFS ist ein Merkle Tree



- Jeder innere Knoten besitzt Hash von seinen Kind Knoten.
- Hash 0-0 und 0-1 sind die Hashes für Data Blocks 0 und 1
- Hash 0 ist der Hash der Verkettung der Hashes 0-0 und 0-1
- Verifikation eines Knotens in Zeit:  $O \log(n)$
- Merkle Signatur Schema braucht nur ein public Key für  $2^n$  Knoten Signaturen

# Block Pointer



- DVA = data virtual address
- DVAs zeigen bis auf 3 verschiedene Blöcke mit gleichen Daten:
  - DVA für Daten
  - DVAs für File System Meta Data
  - DVAs für global Metadata über alle File System Instanzen im Pool.

# Inhalt

- Buffer Cache
- Szenarien Auffinden eines Buffers.
- Virtual File System (VFS)
- moderne Filesysteme
- ZFS Struktur
- Copy on Write
- Snapshots
- RAID
- Skalierbarkeit



# Erinnern Sie sich?



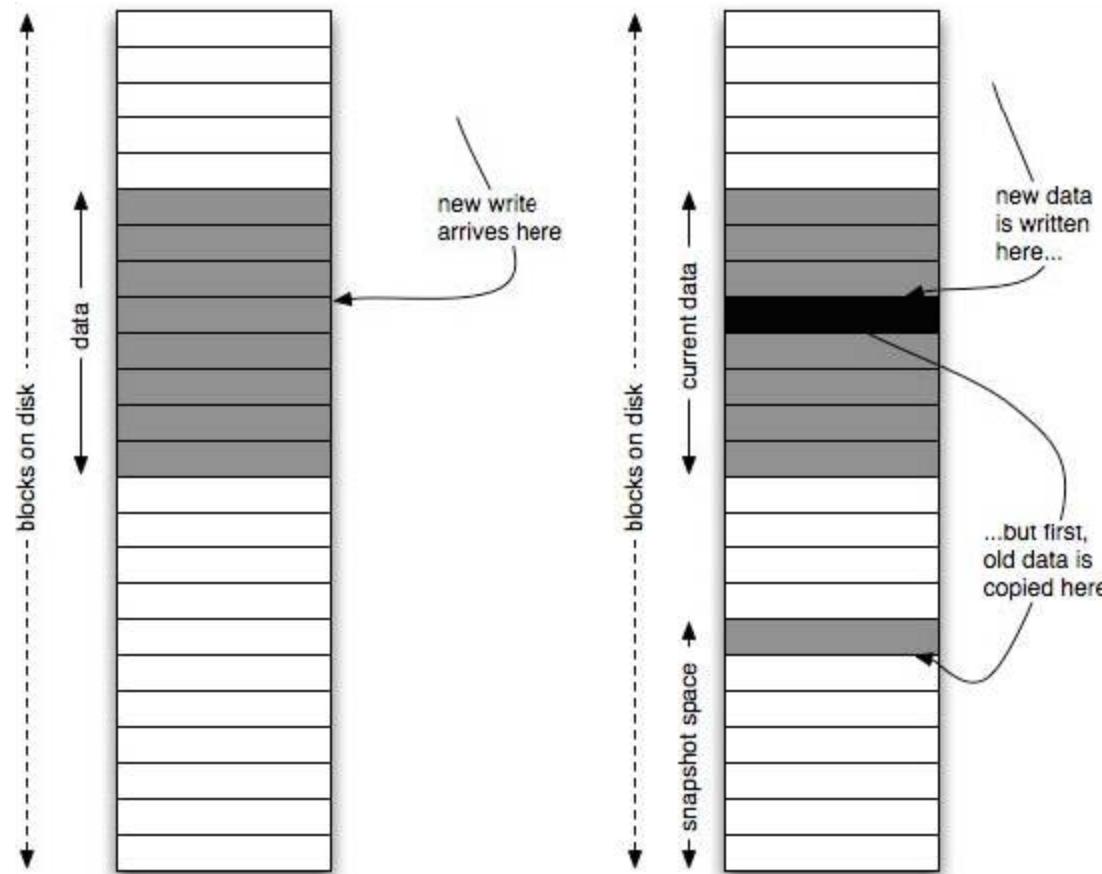
## Konsistenz von Filesystemen

- ◆ Nach einem Systemzusammenbruch kann das Filesystem in einem inkonsistenten Zustand sein, insbesondere, wenn Blocks für die Dateiverwaltung (I-Nodes, Liste freier Blocks, Verzeichnisse) nicht korrekt auf die Festplatte geschrieben wurden.
- ◆ Hilfsmittel: scandisk, fsck
- ◆ Der Konsistenztest erfolgt auf File- und Blockebene:
  - ◆ Bestimmte Blöcke sind weder Files zugeordnet noch frei
  - ◆ Ein Block tritt zweifach als frei auf
  - ◆ Ein Block ist zwei verschiedenen Files zugeordnet

# Konsistenz der Daten

- `fsck` angewendet auf kann von einigen Minuten bis zu 72 Stunden (sehr grosse Filesysteme) in Anspruch nehmen.
- Wir möchten zu jederzeit wissen wenn Daten Korrupt sind, nicht erst beim Verwenden der Daten.
- Vergl. spätere Folie „Selbstheilendes ZFS“
- *CoW* „Copy on Write“ und *RoW* „Redirect on Write“ werden angewandt um jederzeit konsistent zu sein.

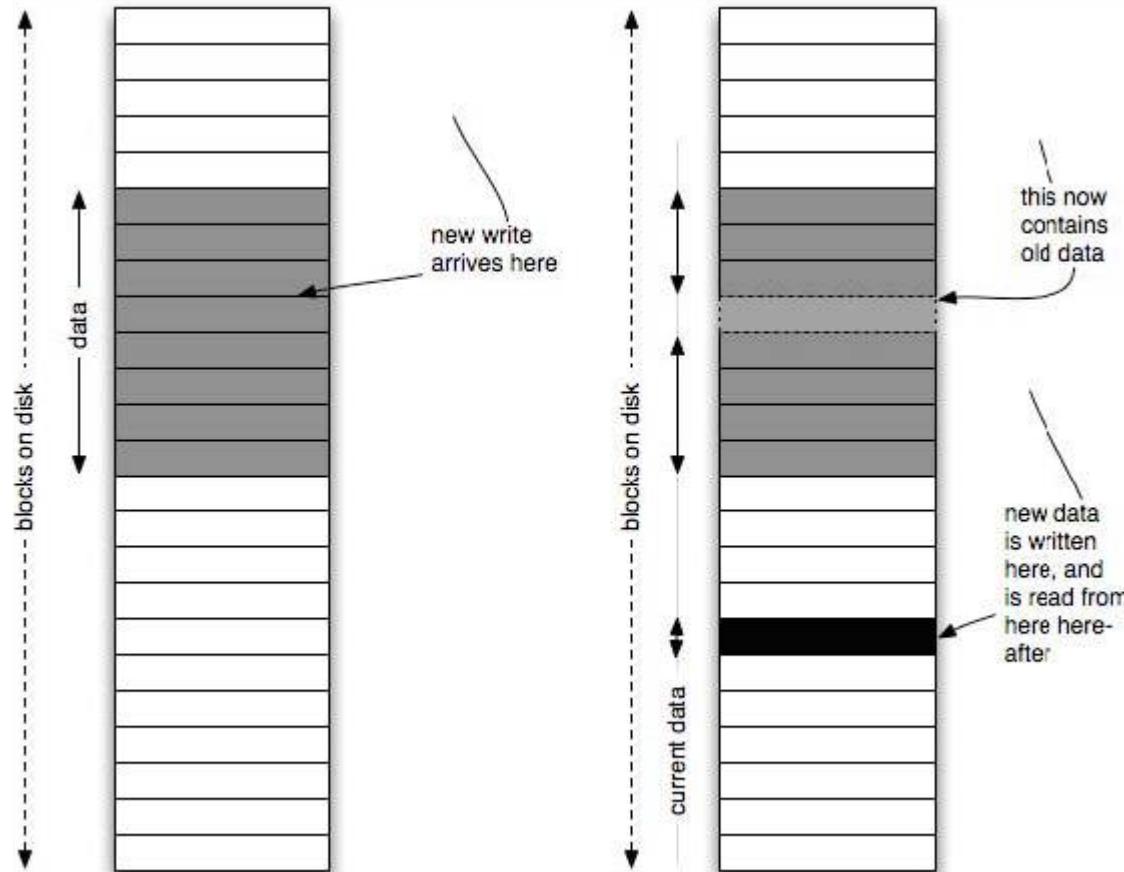
# CoW (copy on write)



Die alten Daten werden in einen neuen Speicherblock kopiert.

Die Daten im alten Speicherblock werden überschrieben mit den neuen Daten.

# RoW (redirect on write)



Neue Schreibanfrage möchte die Daten in den alten Block schreiben.

Diese werden aber in einen neuen Block geschrieben (redirect). Der Originalblock enthält alte Daten.

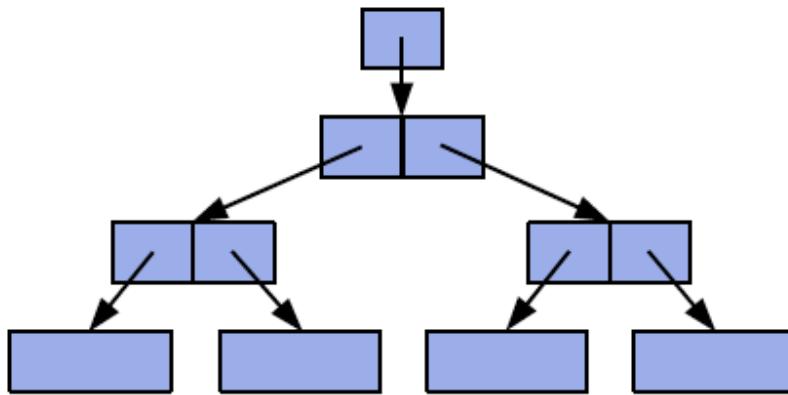
# ZFS CoW ist eher ein RoW

ZFS verwendet ein “reallocate or redirect on write” transactional object model. Alle Block Pointer innerhalb des Filesystems beinhalten eine 256-Bit Checksumme vom Ziel-Block welcher verifiziert wird beim Lesen.

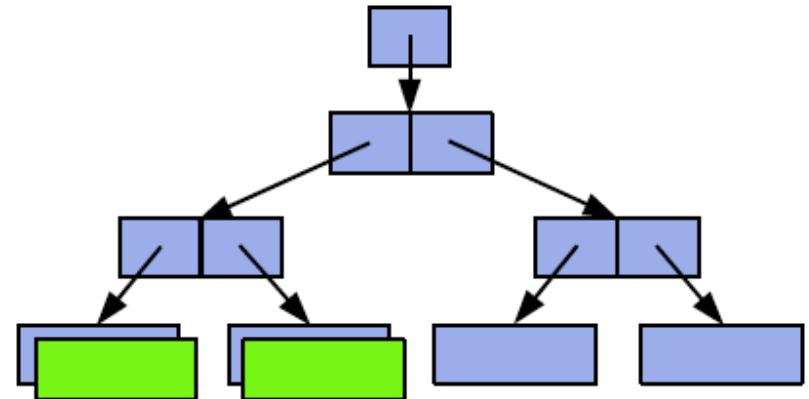
Blöcke mit aktiven Daten werden nie überschrieben. Ein neuer Block (Segment) wird alloziert und mit den modifizierten Daten beschrieben. Ebenso jegliche Metadaten-Blöcke die darauf referenzieren werden gleich gelesen, neu alloziert und beschrieben. Um den Overhead dieses Vorgangs zu minimieren werden mehrere Updates in transaktionale Gruppen zusammengefasst. Der Intent Log (ZIL) wird verwendet wenn synchrone Schreib Semantik verlangt wird.

# Copy on Write (CoW)

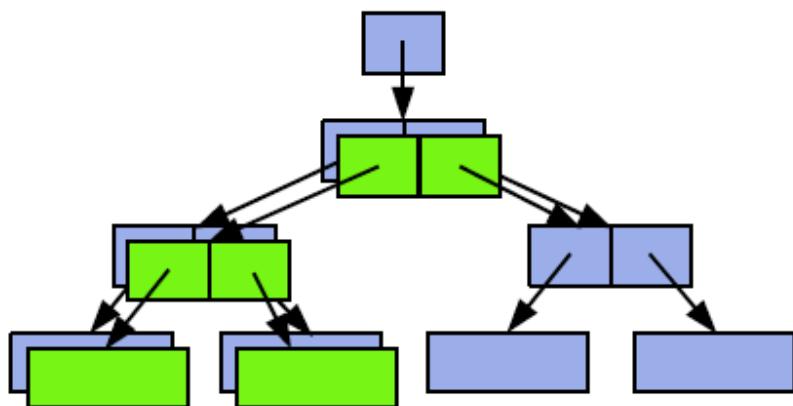
1. Initial block tree



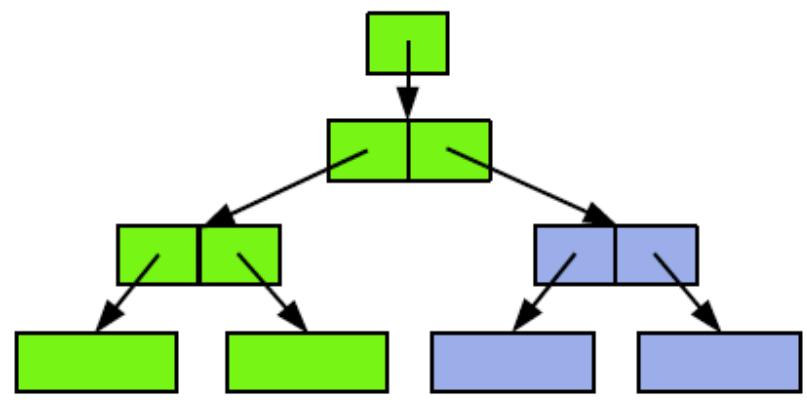
2. CoW some blocks



3. CoW indirect blocks



4. Rewrite uberblock (atomic)

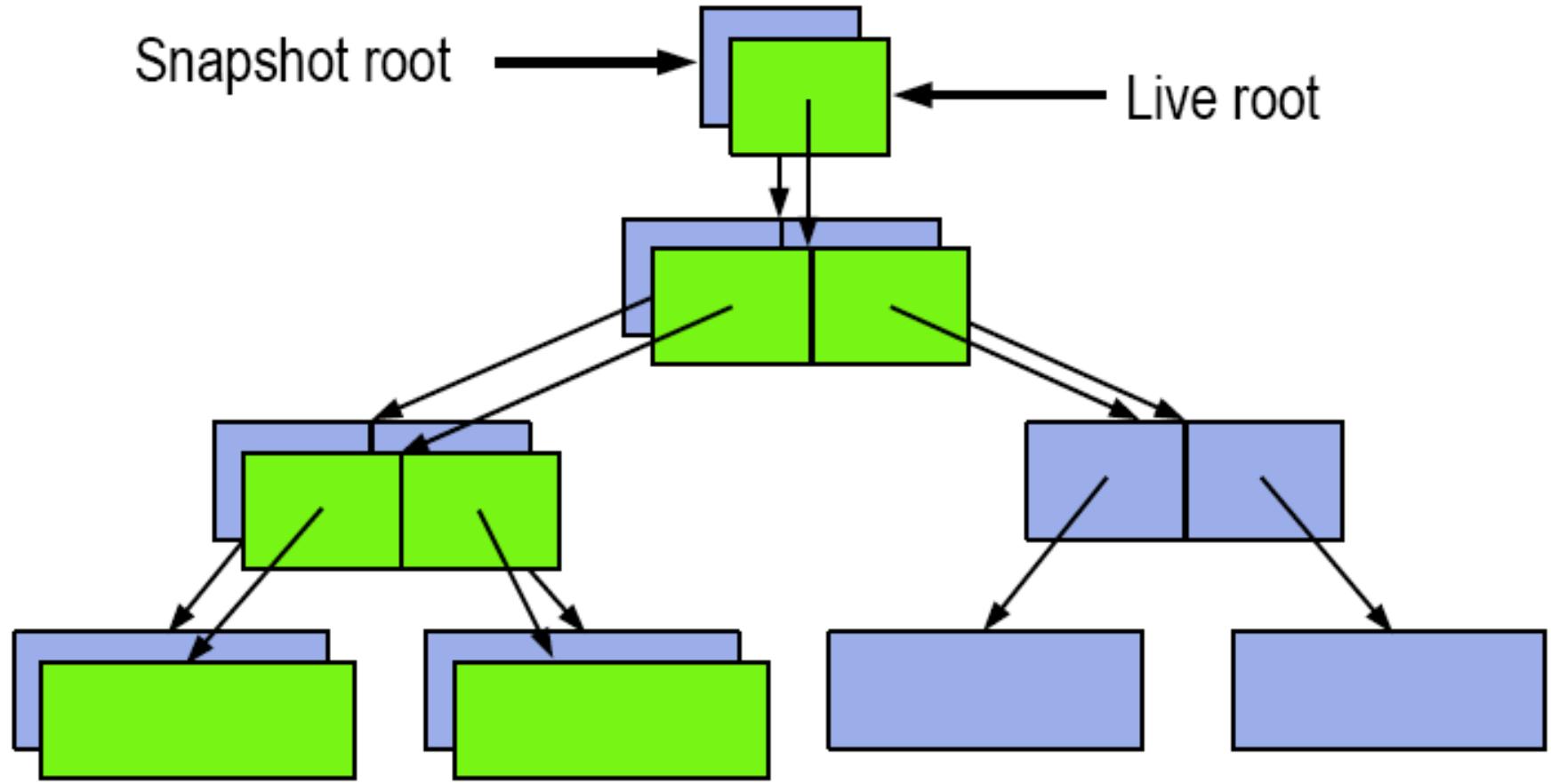


# Inhalt

- Buffer Cache
- Virtual File System (VFS)
- moderne Filesysteme
- ZFS Struktur
- Copy on Write
- Snapshots
- RAID
- Skalierbarkeit

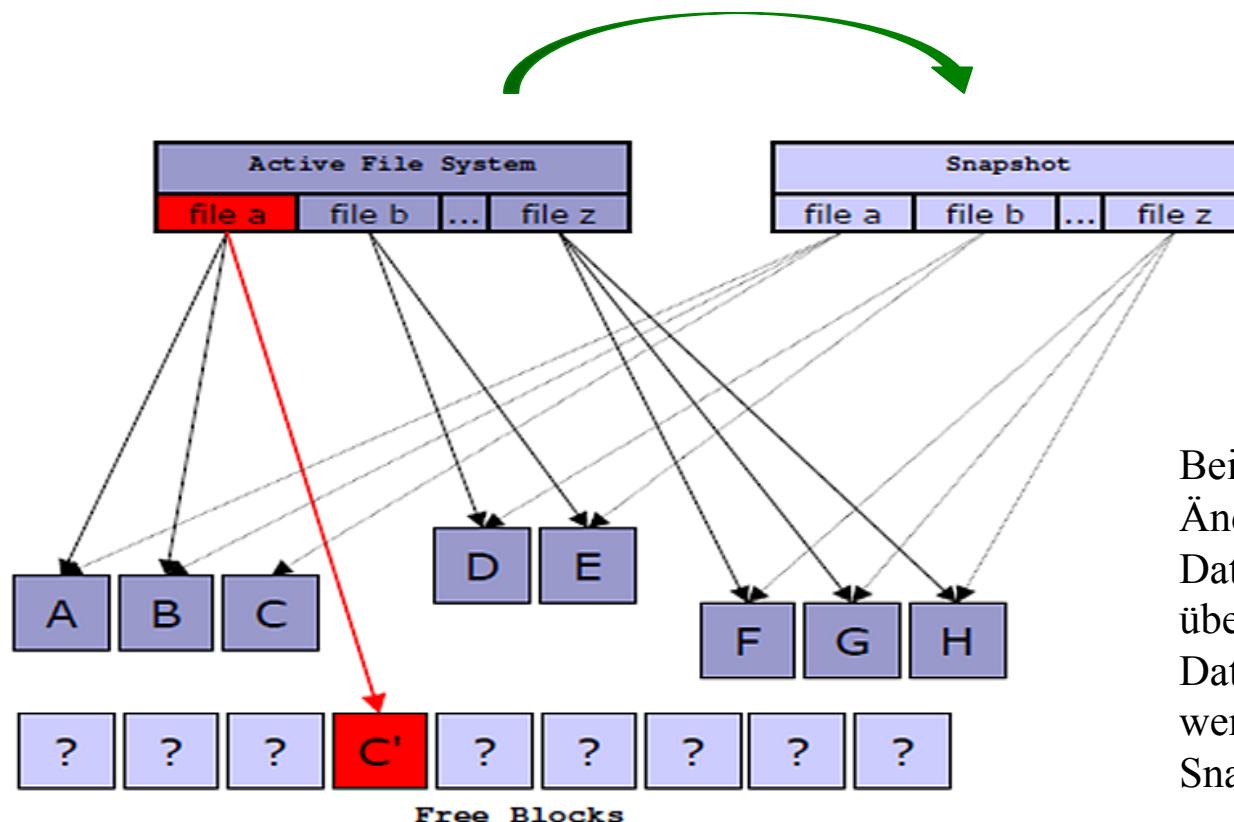


# Konstante Snapshots



Bonus: Es ist einfacher konstante Snapshots zu ziehen

# Snapshot



Bei File System  
Änderungen werden nie  
Datenblöcke  
überschrieben. Alte  
Datenblöcke (freed)  
werden weiter von den  
Schnapsots referenziert.

File a besteht neu aus Block A, B, C'

# Snapshot != Backup

- Snapshots sind keine Backups...
- ...aber sie eignen sich gut zum Restore\*.

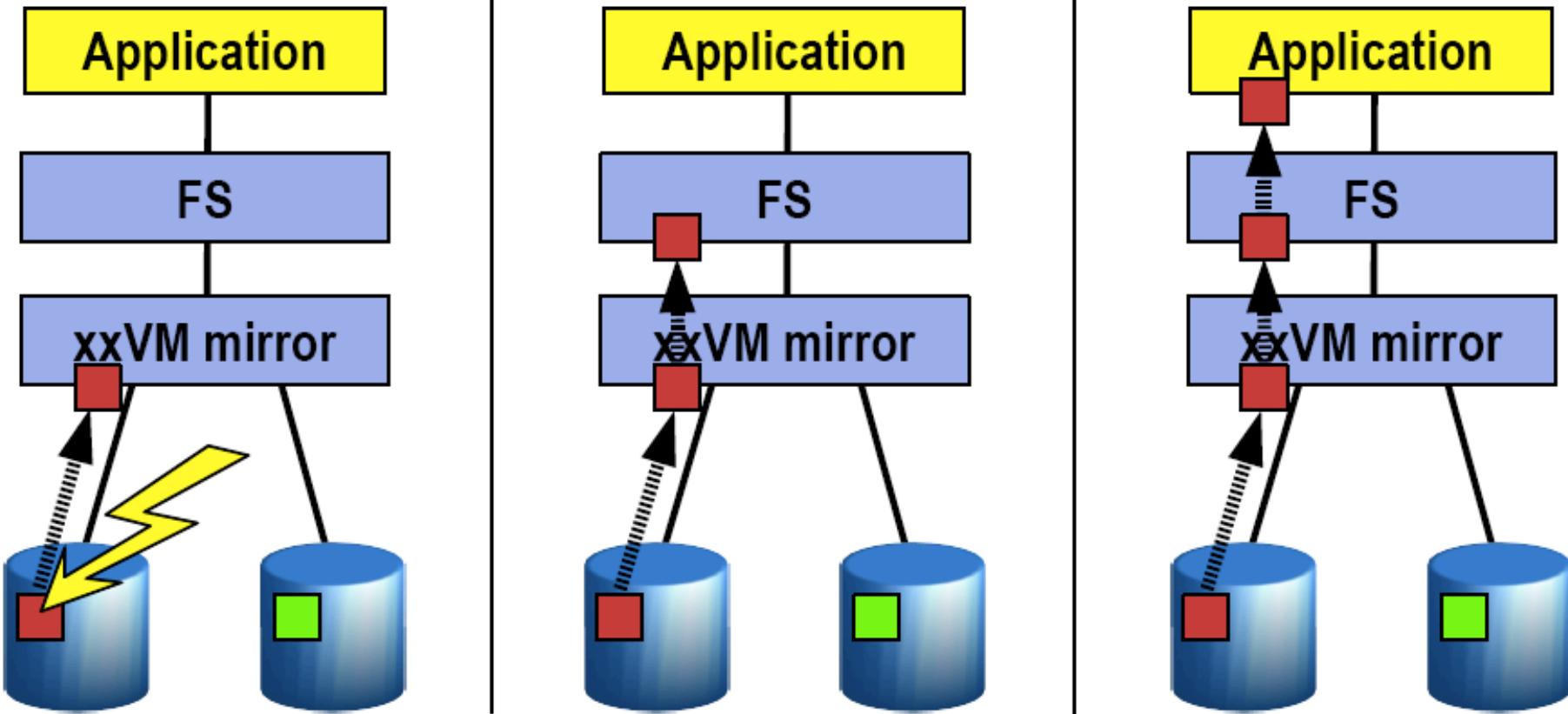
\*will heissen: Von Snapshots kann man sehr einfach einen älteren Zustand wieder herstellen.

# Inhalt

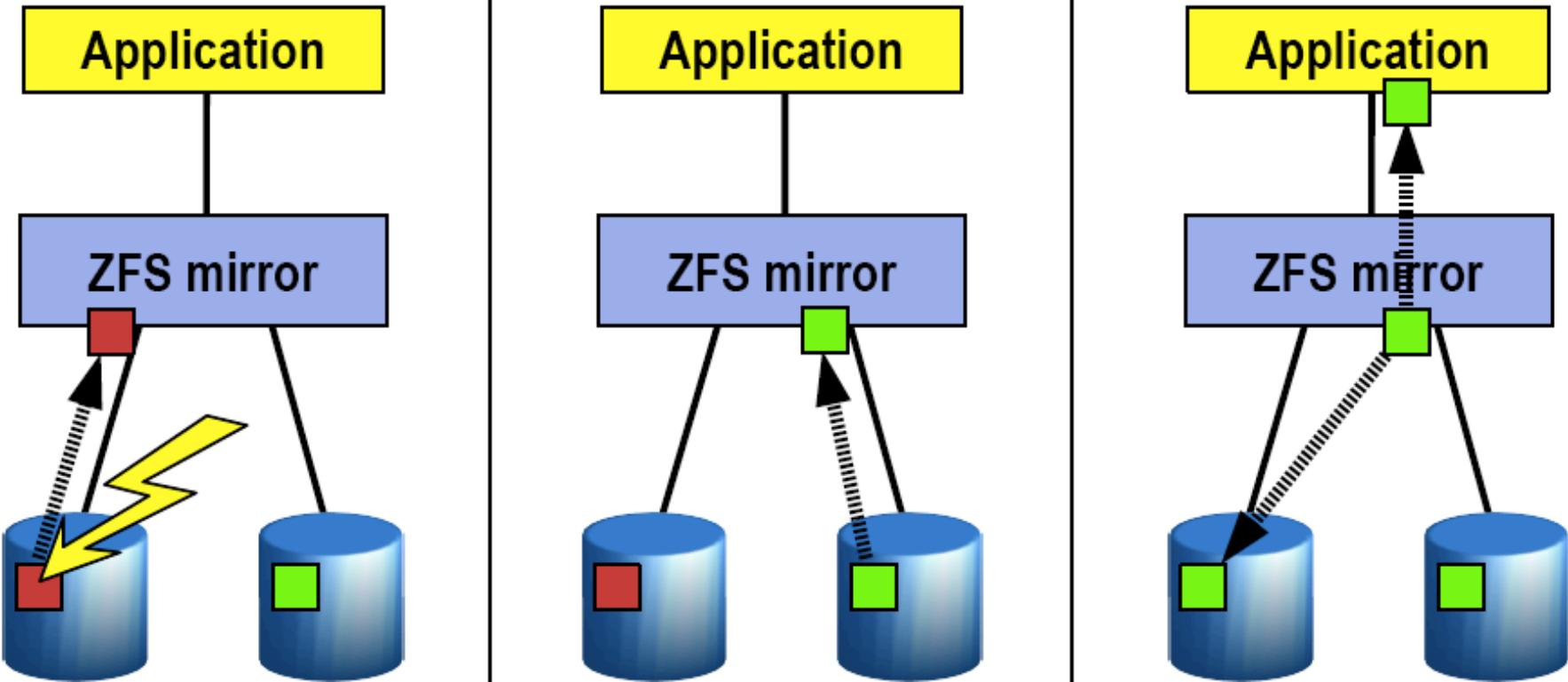
- Buffer Cache
- Virtual File System (VFS)
- moderne Filesysteme
- ZFS Struktur
- Copy on Write
- Snapshots
- RAID
- Skalierbarkeit



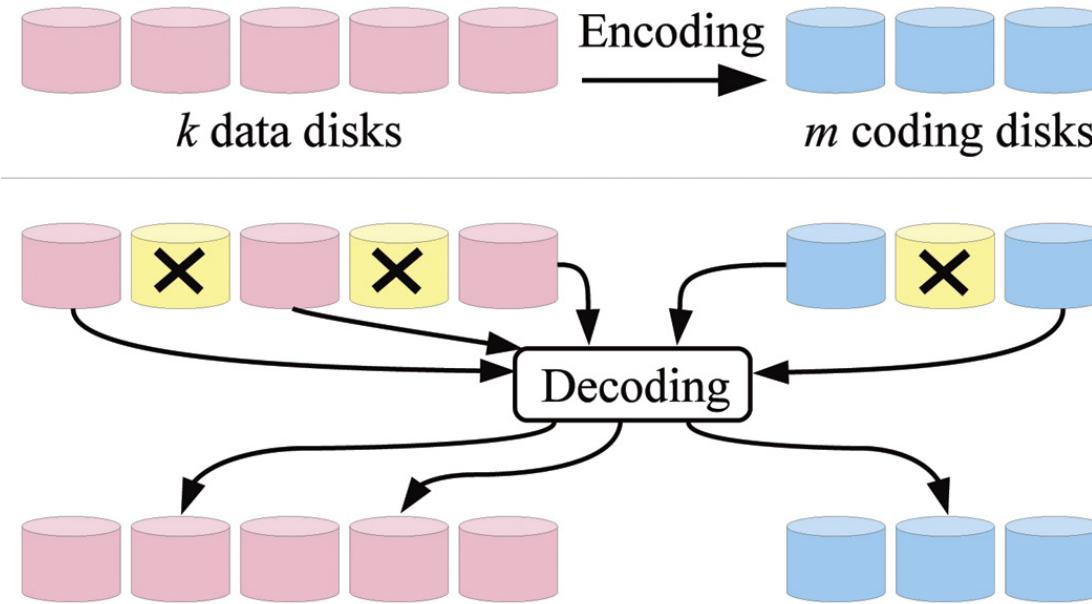
# Traditionelles Mirroring



# Selbstheilendes ZFS



# Erasures Code

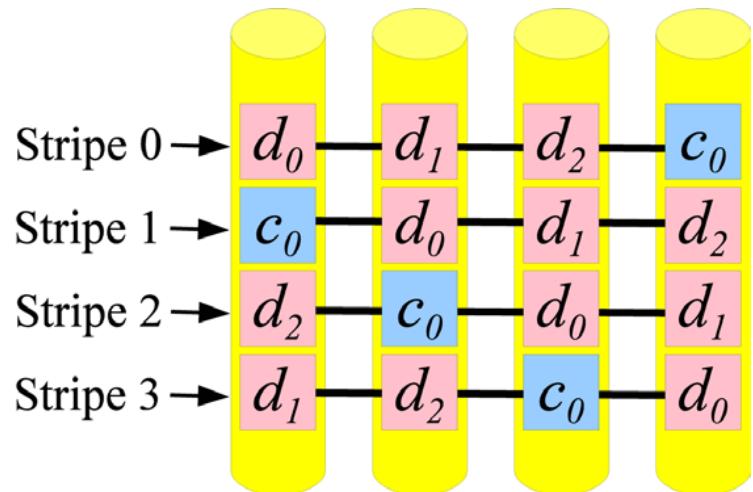


„Erasures Code“ encodes  $k$  Data Disks in  $m$  coding Disks. Wenn  $m$  Disks ausfallen, werden deren Daten vom „erasure Code“ wieder hergestellt.

# Layout von Striping (2 Beispiele)

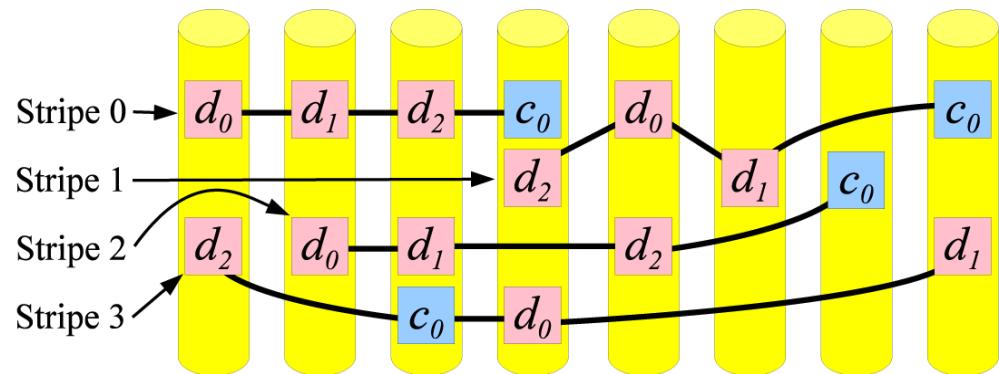
$n = 4$  Disk.

Jeder Stripe:  $k = 3$  Data Stripes  
und  $m = 1$  Coding



$n = 8$  Disk.

Jeder Stripe:  $k = 3$  Data Stripes  
und  $m = 1$  Coding



Weiterführende Literatur siehe: [3]

# Traditionelle RAID 4 und RAID 5

Mehrere Data Disk und eine Parity Disk



Fatal: Fehler beim Schreiben der Parity.  
Bekannt als: *RAID write hole*

Ein kurzer Stromausfall beim  
Schreiben zwischen Data und  
Parity resultiert in „silent data  
corruption“



# RAID Z

- Variable Block Size 512B – 128K
- Single und Double Parity
- Detektiert Silent Data Corruption
- Checksummen getriebene Kombinatorische Rekonstruktion
- Schützt vor mehreren Block Ausfällen wenn nicht reduziert
- Verträgt den Ausfall bis zu 2 Disks
- Überlebt den Ausfall eines defekten Blocks wenn reduziert.

# Inhalt

- Buffer Cache
- Virtual File System (VFS)
- moderne Filesysteme
- ZFS Struktur
- Copy on Write
- Snapshots
- RAID
- Skalierbarkeit



# „Das Filesystem“

- Kann mit allen Fehler-Klassen umgehen
  - Bit rot
  - Phantom writes
  - Misdirected read and writes
  - Administrative errors
- Disk Scrubbing (niemand kann gelöschte Daten lesen)
- Resilvering (rebuilding, reconstructing datas before use)\*
- Real time remote replication (zfs send, zfs receive)
- encryption
- Data deduplication
- ....vieles mehr

\* Marketing Schlagworte wie „selbstheilend“ sind so korrekter beschrieben.

# ZFS Skalierbarkeit

- Wahnsinnige Kapazität (128 Bit)
- Moore's Law sagt: Wir brauchen 65. Bit in 10 – 15 Jahren
- ZFS Kapazität: 256 Quadrillionen ZB (1 ZB = 1 Milliarde Terrabyte)
- Übersteigt die gesamte Speicherkapazität dieser Erde.
- Die Kapazität von ZFS ist so ausgelegt, dass sie *für immer* ausreicht. (wikipedia)

# ZFS Skalierbarkeit

Wortlänge	128 Bit
Volumemanager	Integriert
Ausfallsicherheit	RAID 1, RAID-Z1 (1 Parity-Bit, ~RAID 5), RAID-Z2 (2 Parity-Bits, ~RAID 6) und RAID-Z3 (3 Parity-Bits) integriert
maximale Größe des Dateisystems	$16 \text{ EiB} (= 16 \times 2^{60} \text{ Byte})$
maximale Anzahl an Dateien in einem Dateisystem	$2^{48}$
maximale Größe einer Datei	$16 \text{ EiB} (= 16 \times 2^{60} \text{ Byte})$
maximale Größe jedes Pools	$3 \times 10^{23} \text{ PiB}$ (ca. $2 \times 10^{38} \text{ Byte}$ )
maximale Anzahl an Dateien in einem Verzeichnis	$2^{48}$
maximale Anzahl an Geräten im Pool	$2^{64}$
maximale Anzahl an Dateisystemen im Pool	$2^{64}$
maximale Anzahl an Pools im System	$2^{64}$

# Ein Speicher für die Ewigkeit?

Zur theoretischen Kapazität von ZFS kursiert folgendes Zitat:

*„Ein 128-Bit-Dateisystem zu füllen würde die quantenmechanische Grenze irdischer Datenspeicherung übersteigen. Man könnte einen 128-Bit-Speicher-Pool nicht füllen, ohne die Ozeane zu verdampfen.“* (Jeff Bonwick, Chefentwickler von ZFS)

Zum Verständnis des Zitats sei angemerkt, dass die Speicherung oder Übertragung einer Informationseinheit – z. B. ein Bit – an die Speicherung oder Übertragung von Energie gekoppelt ist, da Information ohne ein Medium nicht existieren kann, d. h. Information ist an die Existenz unterscheidbarer Zustände gekoppelt. Die Gesetze der Quantenmechanik erzwingen eine Mindestmenge von Energie pro Informationseinheit, da die Information sonst aufgrund der quantenmechanischen Unschärfe verlorengeht. Um einen Speicherpool mit 128-Bit-Adressierung zu füllen, wäre eine Energiemenge notwendig, die größer ist als die Menge an Energie, die ausreichen würde, um die irdischen Ozeane zu verdampfen. Gleichzeitig ist „boiling the ocean“ im Englischen ein idiomatischer Ausdruck dafür, etwas Unmögliches zu versuchen. Bonwick illustriert damit, dass ZFS für alle Zukunft genügend Kapazität bietet. (Auszug aus Wikipedia)

# Referenzen

- [1] Maurice J. Bach - The Design of the UNIX Operating System
- [2] S.R. Kleinmann - vNodes: An Architecture for Multiple File System Types in Sun UNIX
- [3] H. P. Anvin. The mathematics of RAID-6:  
<http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf>, 2009.
- [4] Reed-Solomon: <http://de.wikipedia.org/wiki/Reed-Solomon-Code>

Lucerne University of  
Applied Sciences and Arts

**HOCHSCHULE  
LUZERN**

Engineering & Architecture



Questions?