

**Aktuelle Entwicklungen
bei
(Unix-) Linux-
Dateisystemen**

Klaus Kusche, Juni 2015

Inhalt

- **Ziele & Voraussetzungen**
- **“Normale” Filesysteme**
- **Der Unterbau: Block Devices**
- **Filesysteme über's Netz**
- **Filesysteme für Spezialzwecke**

Ziele

- Überblick über die Möglichkeiten und Features der aktuellen Filesysteme in Linux
 - Kenntnis ihrer Vor- und Nachteile
 - Kenntnis typischer Einsatzgebiete
 - Teilweise: Einblick in ihre technische Realisierung
-
- *Möglichkeiten moderner Filesysteme verstehen und sinnvoll nutzen können*
 - *Für gegebene Anforderungen eine gute Storage-Architektur entwerfen können*

Voraussetzungen

Grundkenntnisse:

- Betriebssysteme
- Hardware: Platten usw.
- Netzwerk, im Besonderen SAN

Konkret bezüglich Filesystemen:

Unix UFS/FFS oder Linux ext2

- Rudimentäres Verständnis der Features aus Benutzersicht
- Rudimentäres Verständnis der Implementierung (Daten / Metadaten, Platzverwaltung, ...)

“Normale” Dateisysteme

- Allgemeine Features
- Journaling
(ext3/4, Xfs, Jfs, ...)
- Log-strukturierte Filesysteme
(Btrfs, Nilfs, ...)

Allgemeine Features: ACL's

Statt **rw-rw-rw** Permissions:

Access Control Lists (“*ACL's*”)
pro File/Dir

= Beliebig lange Liste von Paaren:

“Benutzer/Gruppe: Recht”

3 verschiedene de-facto-Standards:

- POSIX nicht-Standard
- SELinux, ...
- NFSv4

Allgemeine Features: Xattr

Verallgemeinerung:

Extended Attributes (“*Xattr*”)
pro File/Dir

= Beliebige Liste von Attributen:
“Name = Wert”

Vom Usermode aus beliebig lesbar / setzbar

Reservierte Xattr vom Kernel genutzt (ACL's, PaX, ...)

Allgemeine Features: Quotas

Quota = Limit

*pro einzelner User / pro Gruppe
für die maximale Platzbelegung
im Filesystem*

Allgemeine Features: Extents

In Platzverwaltungs-Metadaten:

Statt 1 Eintrag pro einzelnem Disk Block (UFS, FAT, ...):

Extent = “*y Blöcke ab Block x*” = 1 Eintrag pro
zusammenhängendem Disk-Block-Bereich

Ziel daher: Speicherung der Daten in möglichst
wenigen, großen, zusammenhängenden Bereichen

Vorteile:

- Weniger Fragmentierung, phys. schnellerer Zugriff
- Weniger Platzbedarf, weniger I/O-Aufwand für die Metadaten (früher viele MB bei großen Files!)

Allgemeine Features: Allokation

Ziel: Möglichst *wenig Extents*
= *Weniger Fragmentierung*

Standard-Trick:

Verzögerte Allokation

erst beim Cache Flush auf Disk

statt bei jedem logischen Schreiben eines Blocks

=> Anzahl der neu zu schreibenden Blöcke abzählbar

=> Alle neuen Blöcke eines Files in einem Extent anlegen!

Weiters: Eigener Sys-Call für explizite Vorab-Allokation

wenn die Applikation die Größe des Files vorab kennt

Allgemeine Features: Große Dir's

Problem: Z.B. Mail-Verzeichnisse:

Können zehntausende Files pro Dir enthalten!

Ziel: *Performance-Verbesserung*

Trees / Hashtables

oder Indices

für Directories

Allgemeine Features: Trim

Das Filesystem meldet
logisch freiwerdenden Platz
an die darunterliegenden Treiber

(die Treiber & Devices wüssten sonst nicht,
welche Blöcke aktuell unbenutzt / unnötig sind)

==> Großer Vorteil für die interne Verwaltung von **SSD's**:
Höhere Geschwindigkeit, viel **größere Lebensdauer**

==> **Weniger Platzverbrauch**
auf virtuellen Storages von VM's und in Clouds
und auf Storage-Servern mit "Thin Provisioning"

Journaling: Motivation

Der Filesystem Check nach Crash usw.

- ... dauert bei großen FS “**ewig**” (viele Stunden!):
Liest alle Metadaten (Inodes, Dir’s, Allocation Maps, ...) !
- ... kann **schiefgehen** ==> FS irreparabel kaputt
(Grund: “Halb” geschriebene Metadaten-Updates)
- ... prüft nur Metadaten
==> korrumpierte Daten werden trotz “FS ok” nicht erkannt!

Wunsch: Filesystem Check

- ... geht “**sofort**” (< 1 Sekunde, Standzeit = Kosten!)
- ... ist immer **erfolgreich**, d.h. Metadaten (+ Daten) ok

Journaling: Idee

(am Beispiel ext3 / ext4, ähnlich in Xfs, Jfs, ...):

- Jedes Filesystem bekommt ein **“Journal”**:
 - Eigener, fix reservierter Platz fixer Größe (einige MB)
 - Kann in selber oder auf separater Partition liegen
 - Wird zyklisch sequentiell beschrieben
- Zusammengehörende Metadaten-Updates einer logischen Operation (z.B. Dir + Inode + Allokation Map) werden zu logischen **“Transaktionen”** zusammengefasst
- **Ziel**: Zum FS Check reicht das **Lesen des Journals!**

Journaling: Ablauf

Bei Einstellung **data=ordered** (default, mittlerer Schutz):

- Daten-Änderungen ins Filesystem schreiben
- **Write barrier** oder Cache flush / Disk sync
- Dazugehörige Metadaten-Updates ins Journal schreiben, als eine zusammengehörige Transaktion markieren, Status der Transaktion: **“offen”**
- **Write barrier** oder Cache flush / Disk sync
- Dieselben Metadaten-Updates im Filesystem machen
- **Write barrier** oder Cache flush / Disk sync
- Transaktion im Journal als **“fertig”** markieren

Journaling: FS-Check

Konsistenz der Metadaten beim FS-Check:

- Transaktion nicht oder nicht vollständig im Journal:
=> Noch nichts im FS, FS hat garantiert noch konsistenten Zustand vor der Änderung
- Transaktion vollständig im Journal, Status "offen":
Update kann nicht, halb oder komplett im FS sein?!
=> Gesamte Transaktion aus dem Journal lesen
und im FS nochmal durchführen
=> FS hat danach konsistenten neuen Zustand
- Transaktion im Journal als "fertig" markiert
=> Transaktion sicher komplett im FS, FS konsistent neu

Journaling: Daten-Konsistenz

- **data=ordered:**
Daten sind ev. jünger als Metadaten, aber nie älter
(vor jedem Metadaten-Update
sind die dazugehörigen Daten schon fertig im FS)
- **data=journal:**
Daten werden wie Metadaten ins Journal geschrieben
==> Viel langsamer, schreibt alle Daten doppelt
==> Daten und Metadaten garantiert 100 % konsistent
- **data=writeback:** Kein flush / sync nach den Daten
==> Etwas schneller
==> Ev. Metadaten-Update fertig,
aber Daten dazu fehlen noch!

Journaling: Probleme

- Alle Metadaten (ev. auch alle Daten) werden doppelt geschrieben
 - Weniger Parallelität, mehr Cache Flushes, öfter warten bis alles wirklich auf Disk geschrieben ist
- ==> Schlecht für Performance (vor allem bei SSD's)
- ==> Schlecht für SSD-Lebensdauer

Daher auf SSD:

Ev. Filesystem ohne Journaling verwenden!

- ext4 + **data=ordered** + delayed alloc = “*rename bug*” (fixed)

Log-strukturierte FS: Idee

Ganz grob: Das ganze FS wird

ähnlich wie ein Journal verwaltet

- Es wird immer nur sequentiell neu geschrieben (in bisher freien Platz), nie "in Place" geändert
- Statt Ändern (z.B. von Metadaten):

"copy on write"

- Geänderte Daten in neuen Block schreiben
- Verweis darauf von alten auf neuen Block ändern (ebenfalls wieder durch Neuschreiben)
- Alten Block freigeben

Log-strukturierte FS: Effekt

Daten werden vor den Metadaten geschrieben,
dann die Metadaten “von unten nach oben”
und zuletzt ein neuer “Wurzelblock” gespeichert
(“Wurzelblock” = oberster Ausgangspunkt aller Metadaten,
Beispiel BtrFS: Speichert intern alles in einer Art B-Trees)

=> Solange kein neuer “Wurzelblock” geschrieben:
Am zuletzt gültigen Wurzelblock hängt
der alte, unveränderte, konsistente Altstand,
alle Änderungen danach sind noch “unsichtbar”

=> Wenn neuer “Wurzelblock” fertig:
Alle Änderungen dieser Transaktion sind erledigt,
konsistenter neuer Stand wird sichtbar

Log-strukturierte FS: Snapshots, ...

Wichtigster Vorteil von Log-strukturierten FS:

Snapshots und ***File/Dir Versioning*** sind trivial
(Einfrieren und Aufheben alter “Momentaufnahmen”
des gesamten Filesystems oder einzelner Files/Dirs):

Einfach alten (Wurzel-) Block mit allem was dranhängt
aufheben statt freigeben

- Erstellung “sofort” möglich, kein Aufwand
- Nur seitdem geänderte Daten brauchen doppelt Platz,
gleiche Daten sind nur einmal gespeichert
- Alte Snapshots sind sogar read/write mount-bar
- Neue File-Operation “clone File” bzw. “clone Dir”

Log-strukturierte FS: Nutzen

Anwendung von Snapshots:

- **Konsistente Backups** bei “lebendem” System (ohne Unterbrechung des Anwendungsbetriebs)
 - Neuen Snapshot zur Backup-Zeit anlegen (geht “blitzartig”)
 - Anwendungen arbeiten am Haupt-FS weiter
 - Backup-Software kann parallel dazu in Ruhe den Snapshot-Stand auf Band sichern
 - Beeinflusst sich gegenseitig nicht!
- **“Undelete”**-Feature für vor Kurzem gelöschte Files
- **Rücksetzen** nach experimentellen Updates usw.

Log-strukturierte FS: Vorteile

Weitere Vorteile:

- Filesystem Check nach Crash ist relativ trivial:
Nur letzten gültigen "Wurzelblock" suchen!
(die Wahrheit ist komplizierter...)
- Sollte sich gut für SSD's eignen
(bisher kein merklicher Performance-Gewinn...)
- **BtrFS**: "Eierlegende Wollmilchsau?"
 - Multi-Device-Verwaltung, RAID integriert, ...
 - Komprimierung, Prüfsummen, ...

Log-strukturierte FS: Nachteile

- Mehr Writes
- Mehr Fragmentierung, weniger Lokalität:
Daten / Metadaten werden immer weiter verstreut
=> Zugriff vor allem bei Platten langsamer

Gegenmittel: ***Garbage Collection***

Vereinzelte Daten in Disk-Bereichen mit viel Freiplatz werden unverändert unkopiert, um große, zusammenhängende freie Blöcke zu schaffen

- Extrem hohe algorithmische Komplexität nagt am Vertrauen in den Code...

Der Unterbau: Block Devices

- Grundlagen Block Devices
- Logical Volume Manager
- Loop Devices
- DRBD (Distributed Replicated Block Developes)

Block Devices

Speicher aus der Sicht des Filesystems:

Ein “***Block Device***” pro Filesystem,
z.B. **`/dev/sda`**

= *Lineare, sequentiell durchnummerierte
Folge gleichgroßer Blöcke (512 B - 4 KB)
mit Random Access*

Abstraktion, HW-unabhängig!

Im einfachsten Fall:

1 Block Device = 1 Platten-Partition

Block Device Driver

“Block Device Driver”

liefern diese Abstraktion
aufbauend auf konkreter Storage-HW:

==> Enthalten die HW-abhängigen Treiber

==> Arbeiten nur blockweise

==> Wissen nichts vom FS

==> Können mit beliebigem FS verwendet werden

(und umgekehrt:

Ein FS kann mit beliebiger HW verwendet werden)

“Normale” Block Devices

- Disk, SSD, CD-ROM, Floppy, USB-Stick, ...
 - **SCSI, SATA, SAS, USB Storage Protokoll, NVM, ...**
 - **Proprietäre RAID-Controller**
- **Via Netz:**
 - **Fibre Channel (für SAN + Storage Server)**
 - **iSCSI (SCSI-Plattenprotokoll über Ethernet)**
- **Im RAM**

Logical Volume Manager (1)

In Linux:

DM (Device Mapper)

Grundlegende Features:

- Bietet logische Block Devices größer als ein phys. Laufwerk
- Dynamische Platzverwaltung
(Laufwerke nachträglich dazu,
Platz zwischen Partitionen umverteilen, ...)
- Software RAID 0/1/5/6/10

Logical Volume Manager (2)

Fortgeschrittene Features:

- **Caching** auf Block-Ebene
(optimiert für SSD als Cache für Disks)
- **Verschlüsselung** ganzer Partitionen
- **Snapshots** ganzer Partitionen
- **Thin Provisioning:**
*Nur im FS wirklich belegter Platz
wird auf der realen Hardware allokiert!*
- **Multipath:**
Parallelität und Redundanz bei SAN-Devices

Loop Devices

Der Datenbereich eines normalen Files
(in einem anderen FS)
wird als Block Device abstrahiert

=> Erlaubt *Filesystem innerhalb eines normalen Files*

Anwendung:

- Verschlüsselung (File wird verschlüsselt)
- Filesystem-Images, insbesondere ISO Images von CD's
 - in normalem File erstellen
 - aus einem File mounten

DRBD: Idee

Vorgänger NBD (Network Block Devices):

*“Verlängert” eine Block Device über TCP/IP
von einem Rechner auf einen anderen
(ähnlich iSCSI)*

DRBD:

*Ein logisches DRBD Block Device wird auf
ein lokales und ein oder mehrere nichtlokale
Block Devices **gespiegelt***

==> RAID über mehrere Server verteilt

Kommunikation: TCP/IP oder Infiniband DMA

DRBD: Nutzung (1)

Anwendung:

Hochverfügbare Cluster mit Failover

Normalbetrieb:

- 1 Server hat das FS auf DRBD gemountet, die Applikation läuft auf diesem Server
- DRBD pflegt synchrone Schattenkopien des FS auf allen anderen Servern (dort nicht gemountet)

DRBD: Nutzung (2)

Im Störfall:

(z.B. primärer Server crasht, Cluster bricht)

- Ein anderer Server mountet seine lokale Kopie, übernimmt die Applikations-Ausführung:

Jeder Server kann aktiver Knoten eines Filesystems auf DRBD werden!

- DRBD kann mangels funktionierendem Cluster nicht mehr synchron auf alle Server spiegeln:

DRBD führt daher Buch über alle *lokal geänderten, ungespiegelten Blöcke*
(*modified block bitmap*)

DRBD: Nutzung (3)

Cluster wieder ok:

DRBD zieht die *Änderungen* (und nur diese!)
automatisch *auf allen Servern* nach

==> *RAID wieder synchron*.

(DRBD kommt mit sehr vielen Skripts und Utilities
zur automatischen und händischen Administration)

DRBD: Vorteile

- Viel billiger als SAN und Storage Server (bei im Wesentlichen gleichem Nutzen), basiert auf normalen lokalen Platten:

“Poor Man’s SAN”

- Schneller als jede Cluster-FS-Lösung:
 - “Normales” lokales FS mit viel weniger Overhead
 - Alle Reads lokal (außer bei lokalen I/O-Fehlern)

- Sicherer als normale Cluster-FS-Lösungen:

Alle Daten auf jedem Server, kein single Point of Failure

(Cluster-FS ohne zusätzliches RAID hat keine Daten-Redundanz ==> schützt nur gegen Server-Ausfall, nicht gegen Disk-Ausfall!)

DRBD: Nachteile

- Jedes FS auf DRBD kann zu jedem Zeitpunkt ***nur von einem Server gemountet*** sein
==> **nur aktiv/passiv**-Konfigurationen möglich
- ***Abhilfe: Cluster-FS*** auf DRBD einrichten
==> Erlaubt **aktiv/aktiv**-Betrieb
(ist noch “thin ice”...)

Filesysteme über's Netz

- Unterschied Netzwerk-FS / Cluster-FS
(==> Tafel)
- Netzwerk-FS, NFS v4
- Cluster-FS

Standard-Unix-Netzwerk-FS: NFS

- Seit “ewig”(1985, keine Weiterentwicklung mehr): ***NFS v2 / v3***
 - Diktiert von SUN
 - Stateless Server Design, Protokoll UDP
 - Keinerlei Sicherheit, nicht Firewall-tauglich
- Seit 2000: ***NFS v4*** (seit 2010: ***NFS v4.1***) wird aktiv entwickelt:
 - Offener, hersteller-unabhängiger Standard
 - Komplettes Neu-Design & Rewrite, inkompatibel
 - Stateful Design, Protokoll TCP, Firewall-fähig
 - Kerberos-Integration, verschlüsselt, sehr sicher
 - Skalierbar (seit 4.1): Mehrere Server für ein FS

Andere Netzwerk-FS für Linux

- SMB / CIFS: “*Windows Filesharing*”, Microsoft-bestimmt
Linux kann *Client, Server und Domain Controller* spielen
(Client: Kernel / Server & DC: Usermode: Samba)
- AFS:
 - Für Größt-Installationen (zehntausende Clients)
 - Immer schon Kerberos-basiert
 - Grundideen:
 - Replikation bzw. Komplet-File-Transfers
 - Mehrere / viele Server (“distributed File System”)
 - Alt, sehr eigen, seit Jahren wenig Weiterentwicklung
- ... und ein paar seltene mehr

Netzwerk-FS: Einsatz

*Jedes Linux-Netzwerk-FS
hat Vor- und Nachteile,
keines ist deutlich überlegen,
die Auswahl ist komplex
und Einsatzfall-bezogen!*

Alleinstellungs-Kriterium von NFS:

*Erlaubt **Boot-from-Net** und **diskless Clients***

==> Essentielles Feature

für zentral administrierte Thin-Client-Environments

Cluster-FS: Charakterisierung

- Aus Anwender-Sicht:

Dasselbe Filesystem ist gleichzeitig auf mehreren Servern Read/Write gemountet

- Hardware-seitig:

Mehrere Server greifen gleichzeitig auf Block-Ebene auf dieselben Platten zu (meist via SAN/FC oder iSCSI, ev. Infiniband)

- FS-technisch:

Die Verwaltung eines einzigen Filesystems ist auf mehrere / viele Server verteilt

Cluster-FS: Ziel 1

Hochverfügbarkeit, Ausfallsicherheit

==> HA Cluster (“High Availability”)

- Im Unterschied zu Netzwerk-FS:

***Kein “Single Point of Failure”
auf Server- und Netzwerk-Ebene***

Zusätzlich Plattenspiegelung vorsehen!!!

- Applikationen auf anderen Servern können

***trotz Ausfall eines Servers
unterbrechungsfrei weiterarbeiten!***

Cluster-FS: Ziel 2

Skalierbarkeit, Durchsatz

==> HPC Cluster

(“High Performance Computing”)

- Große Filesysteme (viele TB, ev. PB)
- Kein Bottleneck:
 - Viele parallele Plattenzugriffe
 - Viele parallele Netzwerk-Verbindungen
 - Viele parallele Server (hunderte)
bzw. Knoten eines Supercomputers
 - Lastverteilung

Cluster-FS: Herausforderungen (1)

- Erhalt der File-Locking-Semantik,
Vermeidung kollidierender Zugriffe (parallele Writes)
==> *Distributed Lock Manager*
- Clusterweite (Meta-) Daten-Konsistenz beim Lesen
==> *Cache-Konsistenz-Protokoll* mit Invalidate
- Kollisionsfreie, verteilte Metadaten-Verwaltung
(Directories, Freispeicher-Verwaltung, ...)
- FS-Check (wegen FS-Größe)
==> Entweder Design, das kein FS-Check braucht
==> Oder verteiltes / paralleles FS-Check

Cluster-FS: Herausforderungen (2)

- Design & Implementierung des Cluster-FS darf keinen zentralen Master-Server benötigen, alle müssen gleichberechtigt / allein lebensfähig sein

Sonst: Bottleneck, Single Point of Failure!

- Zusätzlich zu Plattenzugriff via SAN/FC oder iSCSI: Verwaltungs-Protokoll zwischen Servern nötig

(kein Standard, jedes Cluster-FS hat sein eigenes)

- Ev. Server-Quorum zum Schutz des FS bei Ausfällen:

Schreiben darf nur,

wer min. 50 % der Server "sieht":

Vermeidet "Split-Brain-" bzw. "2-Insel-Problem"

Cluster-FS: Freie Realisierungen

- **GFS2** (RedHat)
- **OCFS2** (Oracle)
- **GlusterFS** (RedHat,
kann auch mehrere verteilte RAID-Arten)
- **Lustre** (wichtigstes HPC-Filesystem)
- **Ceph** (HA und HPC, incl. RAID, Snapshots, ...)
- ... und einige proprietäre (IBM GPFS, ...)

Filesysteme für Spezialzwecke

- Filesysteme für spezielle HW
- Union FS, AuFS und Overlay FS
- FS-Cache & Cachefiles
- Usermode-FS (“FUSE”)

Filesysteme für spezielle HW

- ISO-FS usw. für CD und DVD
- JFFS2, LogFS, F2FS, ... für direkt angebundene Flash-Chips (z.B in embedded Systems)
- TmpFs für “RAM-Disks”
- SquashFS:
 - *Read-only*, sonst volle FS-Features
 - *Hoch komprimiert* und Platz-optimiert
 - ==> *Firmware in embedded Systems*
 - ==> Medien für Linux-Life-Systeme
 - ==> Mount-bare Backups und Archive

Union FS, AuFS und Overlay FS (1)

- Vereinigen zwei oder mehr “echte” Filesysteme (ein “oberes” + ein oder mehrere “untere”) zu einem einzigen “virtuellen” Filesystem
- Das “obere” FS hat Priorität, verdeckt das “untere”
=> Wenn derselbe File in beiden FS existiert:
Nur Inhalt und Attribute des “oberen” sichtbar
- **Oberes** FS ist read/write, **unteres** FS ist readonly
=> Alle Änderungen gehen in das obere FS !
=> Spezielle “gelöscht”-Einträge im oberen FS machen Files/Dir’s im unteren FS unsichtbar, ohne das untere FS zu ändern.

Union FS, AuFS und Overlay FS (2)

Anwendungen:

- Änderbare Live-Installationen auf readonly-Medien, z.B.:
“Unteres” FS: Original Linux-Installation auf DVD
“Oberes” FS: Lokale Änderungen und Userdaten auf Stick
- Viele ähnliche Systeme, z.B. in vielen VM's:
“Unteres” FS: Ur-System für alle VM's gemeinsam
“Oberes” FS: Eines pro VM, nur Änderungen ==> klein!
- Experimente mit Konfigurationen, Updates, ...:
“Unteres” FS: Ausgangs-System, bleibt unverändert!
“Oberes” FS: Enthält Änderungen des Experimentes
Wenn Fehlschlag: Oberes FS verwerfen ==> Alter Stand.

FS-Cache und Cachefiles (1)

FScache:

Allgemeine Cache-Logik

für permanenten Cache von beliebigen FS

Arbeitet auf File-Logik-Ebene, nicht auf Disk-Block-Ebene

Spricht den eigentlichen Cache-Speicher
über abstrakte Cache-Backend-Schnittstelle an

Cachefiles:

Backend für FScache:

Nutzt verbleibenden freien Platz in anderem,
normal gemounteten FS zur Speicherung des Caches

FS-Cache und Cachefiles (2)

Übliche Anwendung:

Cache für Netzwerk-File-System in lokalem File-System

Beispiel:

- Root-FS (Linux-System) als ext4 auf lokaler Platte
- Home-FS (Benutzerdaten) remote auf NFS-Server

==> Freier Platz im Root-FS

wird als lokaler Cache für Benutzerdaten genutzt

Ziel: *Performance* (nicht: Disconnected Operation)

Andere Möglichkeiten:

z.B. Cache für **CD/DVD** im Home-FS (**SSD**)

Usermode-FS (“FUSE”)

*Der Linux-Kernel leitet
alle Zugriffe auf das Usermode-FS
an ein normales Usermode-Programm weiter*

==> Die FS-Logik ist *frei implementierbar!*

Anwendungen u.a.:

- Remote FS via WebDAV oder SSH/Scp
- Archiv-Files (.tar, .zip) als FS gemountet
- Spezielle Komprimierungen / Verschlüsselungen
- ...

“The end”

Fragen?