Dateisysteme

Hochleistungs-Ein-/Ausgabe

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Michael Kuhn

2019-04-09

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

Dateisysteme

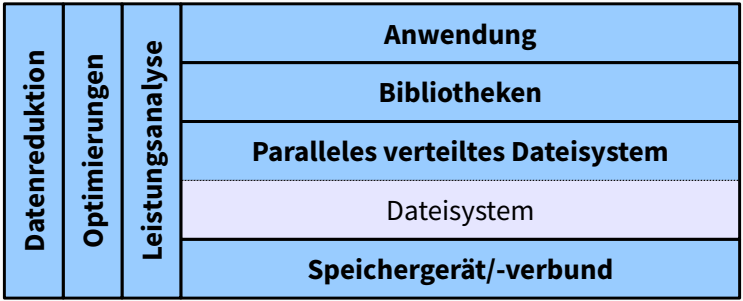


Abbildung 1: E/A-Schichten und orthogonale Themen

Aufgabe

1. Strukturierung
   - Üblicherweise hierarchische Organisation
     - Auf Basis von Dateien und Verzeichnissen
   - Andere Ansätze: Tagging, Queries etc.
2. Verwaltung von Daten und Metadaten
   - Blockallokation und -verwaltung
     - Zugriff über Datei- und Verzeichnisnamen
   - Zugriffsrechte, Zeitstempel etc.

- Dateisysteme nutzen ein darunter liegendes Speichergerät
  - Kann auch durch Speicherverbund bereitgestellt werden
    - Logical Volume Manager (LVM) und/oder mdadm unter Linux

Beispiele

- Linux: tmpfs, ext4, XFS, btrfs, ZFS
- Windows: FAT, exFAT, NTFS
- OS X: HFS+, APFS
- Universal: ISO9660, UDF
- Pseudo: sysfs, proc

Beispiele...

- Netzwerk: NFS, AFS, Samba
- Kryptographisch: EncFS, eCryptfs
- Parallel verteilt: Spectrum Scale, Lustre, OrangeFS, CephFS, GlusterFS

- Setzen häufig auf darunterliegenden Dateisystemen auf

E/A-Schnittstellen

- Anfragen werden über E/A-Schnittstellen realisiert
  - Schnittstellen für unterschiedliche Abstraktionsebenen
  - Weiterleitung an das eigentliche Dateisystem
- Low-Level-Funktionalität
  - POSIX, MPI-IO
- High-Level-Funktionalität
  - HDF, NetCDF, ADIOS

```
1 fd = open("/path/to/file", O_RDWR | O_CREAT | O_TRUNC,
2                            S_IRUSR | S_IWUSR);
3 rv = close(fd);
4 rv = unlink("/path/to/file");
```

Listing 1: E/A über POSIX-Funktionen

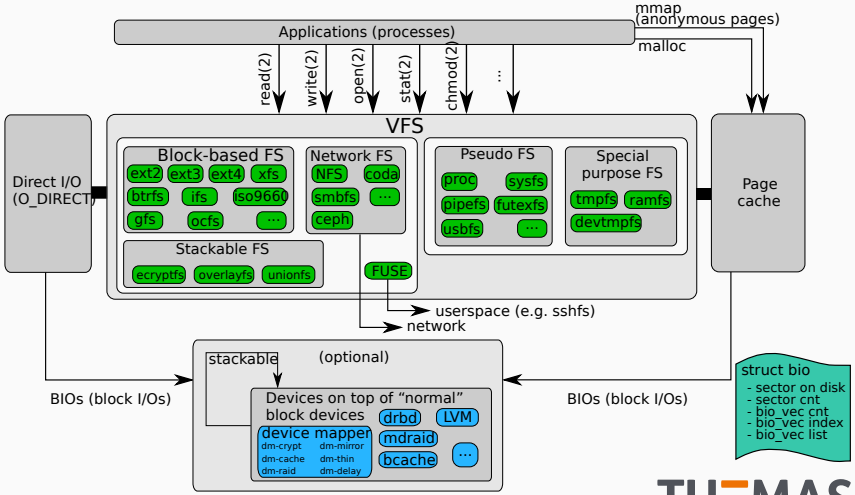
- Mit open können auch Dateien erstellt werden
  - Viele mögliche Flags und Modi
- Initialer Zugriff über Pfad
  - Danach über File Descriptor (bis auf einige Ausnahmen)
- Alle Funktionen liefern einen Rückgabewert
  - Bei Fehlern sollte errno überprüft werden

```
1 nb = write(fd, data, sizeof(data));
```

Listing 2: E/A über POSIX-Funktionen

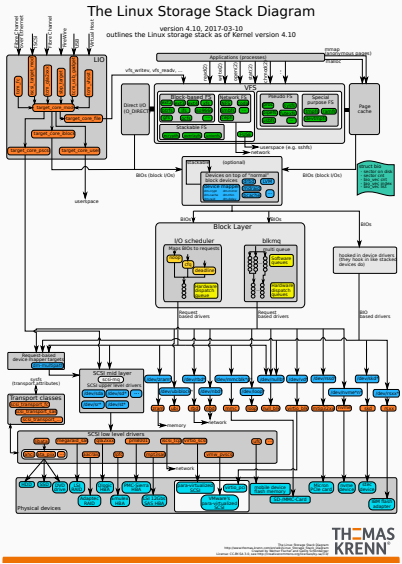
- `write` liefert die Anzahl der geschriebenen Bytes zurück
  - Muss nicht notwendigerweise der übergebenen Größe entsprechen (Fehlerbehandlung!)
  - `write` verändert intern den Dateizeiger (alternativ: `pwrite`)
- Funktionen befinden sich in der `libc`
  - Diese führt System Calls durch

- Virtual File System (Switch)
- Zentrale Dateisystemkomponente im Kernel
  - Standardisiertes Interface für alle Dateisysteme (POSIX)
  - Gibt Dateisystemstruktur und -schnittstelle größtenteils vor
- Leitet Anfragen der Anwendungen an das entsprechende Dateisystem weiter
  - Basierend auf dem Mountpoint
- Ermöglicht die Unterstützung unterschiedlichster Dateisysteme
  - Anwendungen bleiben durch POSIX trotzdem portabel



THOMAS KRENN®

The Linux Storage Stack Diagram
http://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
Created by Werner Fischer and Georg Schönberger
License: CC-BY-SA 3.0, see http://creativecommons.org/licenses/by-sa/3.0/


The Linux Storage Stack Diagram
version 4.10, 2017-03-10
outlines the Linux storage stack as of Kernel version 4.10
applications (processes)
VFS
Page cache
Direct I/O
Stackable FS
Block Layer
I/O scheduler
blkmq
SCSI mid layer
SCSI upper level drivers
SCSI low level drivers
Physical devices
THOMAS KRENN

Dateisystemobjekte

- Unterscheidung in Benutzer- und Systemsicht
  - Benutzer sehen Dateien und Verzeichnisse mit Metadaten
    - Dateien bestehen aus Bytes, Verzeichnisse enthalten Dateien und Verzeichnisse
  - Das System verwaltet Interna
    - Fasst mehrere Blöcke zu Dateien zusammen etc.

- Inodes
  - Eigentliche Basisobjekte des Dateisystems
    - Jeder Datei und jedem Verzeichnis ist ein Inode zugeordnet (siehe `stat`)
  - Enthalten Metadaten
    - Sowohl für Benutzer sichtbare als auch interne
  - Üblicherweise eindeutige IDs und fixe Größe

Dateisystemobjekte...

- Dateien
  - Enthalten Daten in Form eines Byte-Arrays
    - POSIX macht keine weitergehenden Vorgaben
  - Können gelesen/geschrieben werden (explizit)
  - Können in den Speicher gemappt werden (implizit)
- Verzeichnisse
  - Zur Organisation des Namensraumes
    - Können Dateien und weitere Verzeichnisse enthalten
  - Aus Benutzersicht eine Liste
    - Intern häufig Baumstrukturen

```
1 nb = pwrite(fd, data, sizeof(data), 42);
2 nb = pread(fd, data, sizeof(data), 42);
```

Listing 3: Expliziter Zugriff

- `pwrite` und `pread` verhalten sich wie `write` bzw. `read`
  - Explizite Angabe des Offsets und damit threadsicher
- Zugriff über File Descriptor
  - Kann von mehreren Threads parallel genutzt werden

```
1 char* pt = mmap(NULL, file_size, PROT_READ | PROT_WRITE,
2                                  MAP_SHARED, fd, offset);
3 memcpy(pt + 42, data, sizeof(data));
4 memcpy(data, pt + 42, sizeof(data));
5 munmap(pt, FILE_SIZE);
```

Listing 4: Impliziter Zugriff

- mmap erlaubt es eine Datei in den Speicher einzublenden
  - Datei wird an Adresse pt eingeblendet
  - Verschiedene Sichtbarkeitseinstellungen (shared vs. private)
- Zugriff wie auf andere Speicherobjekte
  - Z. B. via memcpy oder direkte Zuweisung

- Beide Zugriffsarten haben jeweils Vor- und Nachteile
  - Beide Modi profitieren vom Caching durch das Betriebssystem
- Expliziter Zugriff
  - Vorteile: genaue Kontrolle über E/A, Möglichkeit direkter E/A
  - Nachteile: separate Puffer notwendig, Kopiervorgänge zwischen Kernel- und Userspace
- Impliziter Zugriff
  - Vorteile: keine separaten Puffer notwendig, effiziente E/A durch das Betriebssystem, keine unnötigen Kopiervorgänge, große Dateien können komplett gemappt werden
  - Nachteile: keine genaue Kontrolle, kompliziertere Fehlerbehandlung (Signale)

Inode	Größe	Namenslänge	Dateityp	Name
23	10	2	2	`.`
24	11	3	2	`..`
⋮	⋮	⋮	⋮	⋮
42	14	6	1	`hello`
42	14	6	1	`world`

Abbildung 2: ext4-Verzeichniseintrag [1]

- Traditionell lineares Array
  - Langsam, da über das komplette Array iteriert werden muss
- Heutzutage eher Baumstrukturen
  - Deutlich komplexer, dafür geringere Zugriffszeiten
- Name wird nicht im Inode gespeichert
  - Mehrere Namen können auf denselben Inode zeigen

Feldgröße	Inhalt
2 Bytes	Berechtigungen
2 Bytes	Benutzer-ID
4 Bytes	Dateigröße
4 Bytes	Zugriffszeit
4 Bytes	Inode-Änderungszeit
4 Bytes	Datenänderungszeit
4 Bytes	Löschzeit
2 Bytes	Gruppen-ID
2 Bytes	Linkzahl
⋮	⋮
60 Bytes	Blockzeiger, Extent-Baum oder Inline-Daten
⋮	⋮
4 Bytes	Versionsnummer
100 Bytes	Freier Speicher

Abbildung 3: ext4-Inode (256 Bytes) [1]

Inodes...

- Kompliziert durch Rückwärtskompatibilität
  - On-Disk-Format kann nur schwer geändert werden
- Viele Felder sind aus Kompatibilitätsgründen aufgeteilt
  - Zeitstempel: 4 Bytes für Sekunden seit 1970, 4 Bytes für Nanosekundenauflösung
  - Größe: Obere und untere 4 Bytes
- Felder mehrfach überladen
  - Blockzeiger, Extent-Baum oder Inline-Daten (falls Datei kleiner als 60 Bytes)
  - 100 Bytes am Inode-Ende für erweiterte Attribute

```
1  $ touch foo
2  $ ls -l foo
3  -rw-r--r--. 1 user group 0 19. Apr 18:48 foo
4  $ ln foo bar
5  $ ls -l foo bar
6  -rw-r--r--. 2 user group 0 19. Apr 18:48 bar
7  -rw-r--r--. 2 user group 0 19. Apr 18:48 foo
8  $ stat --format=%i foo bar
9  641174
10 641174
11 $ rm foo
12 $ ls -l bar
13 -rw-r--r--. 1 user group 0 19. Apr 18:48 bar
```

Listing 5: Inode vs. Datei

POSIX-Schnittstelle

- Syntax beschreibt verfügbare Operationen und deren Parameter
  - open, close, creat
  - read, write, lseek
  - chmod, chown, stat
  - link, unlink
  - (f)truncate, fallocate
- Semantik spezifiziert wie sich E/A-Operationen verhalten sollen
  - `write`: *“POSIX requires that a read(2) which can be proved to occur after a write() has returned returns the new data. Note that not all filesystems are POSIX conforming.”*

- Sparse-Dateien: Dateien mit „Löchern“
  - Z. B. mit `lseek` oder `truncate`
  - Effiziente Speicherung von Dateien mit vielen 0-Bytes

```
1 $ truncate --size=1G dummy
2 $ ls -lh dummy
3 -rw-r--r--. 1 user group 1,0G 18. Apr 23:49 dummy
4 $ du -h dummy
5 0   dummy
```

Listing 6: Erzeugung einer Sparse-Datei

- Preallokation: Speicher vorallokieren
  - Mit `fallocate` bzw. `posix_fallocate`
  - Verhindert Fragmentierung bei vielen Dateivergrößerungen

```
1 $ fallocate --length $((1024 * 1024 * 1024)) dummy
2 $ ls -lh dummy
3 -rw-r--r--. 1 user group 1,0G 19. Apr 19:14 dummy
4 $ du -h dummy
5 1,0G    dummy
```

Listing 7: Preallokation einer Datei

- Standard-Dateisystem in vielen Linux-Distributionen
  - Eingeführt 2006, stabil 2008
  - Vorgänger: ext, ext2, ext3
- Statische Festlegung diverser Parameter bei Dateisystemerzeugung
  - Blockgröße, Dateisystemgröße, Inode-Zahl etc.
  - Einige können nachträglich angepasst werden
- Traditionelles Dateisystem
  - Daten werden direkt geändert (kein Copy on Write)
  - Keine Prüfsummen für Daten und Schnappschüsse
  - Keine sonstigen Komfortfunktionen

- Erstes Dateisystem speziell für Linux
  - Nutzte als erstes Dateisystem die VFS-Schicht
- Inspiriert vom Unix File System (UFS)
- Beseitigte Beschränkungen des MINIX-Dateisystems
  - Dateigrößen bis 2 GB
  - Dateinamen bis 255 Zeichen

- Separate Zeitstempel für Zugriff und Inode-/Datenänderung
- Datenstrukturen für zukünftige Erweiterungen ausgelegt
- Testumgebung für neue VFS-Funktionen
  - Access Control Lists (ACLs)
  - Erweiterte Attribute

- Journaling
  - Erklärung folgt später
- Dateisystemvergrößerung zur Laufzeit
  - Nützlich für LVM-Umgebungen
- H-Baum für größere Verzeichnisse
  - Verkürzt die Suchzeiten im Verzeichnis

- Größere Dateisysteme, Dateien und Verzeichnisse
- Extents
- Preallokation, verzögerte Allokation und verbesserte Multiblockallokation
- Journal-Prüfsummen
- Schnellere Dateisystemüberprüfung
- Nanosekunden-Zeitstempel
- Unterstützung für TRIM

Inhalt	Größe
Padding (Blockgruppe 0)	1.024 Bytes
Superblock	1 Block
Gruppenbeschreibung	m Blöcke
Reservierte GDT-Blöcke	n Blöcke
Daten-Bitmap	1 Block
Inode-Bitmap	1 Block
Inode-Tabelle	k Blöcke
Datenblöcke	l Blöcke

Abbildung 4: ext4-Blockgruppe [1]

- Das Speichergerät ist in mehrere Blockgruppen unterteilt
  - Flexible Blockgruppen fassen mehrere Blockgruppen zusammen
- Blockgröße bestimmt Anzahl an Inodes und Datenblöcken pro Blockgruppe

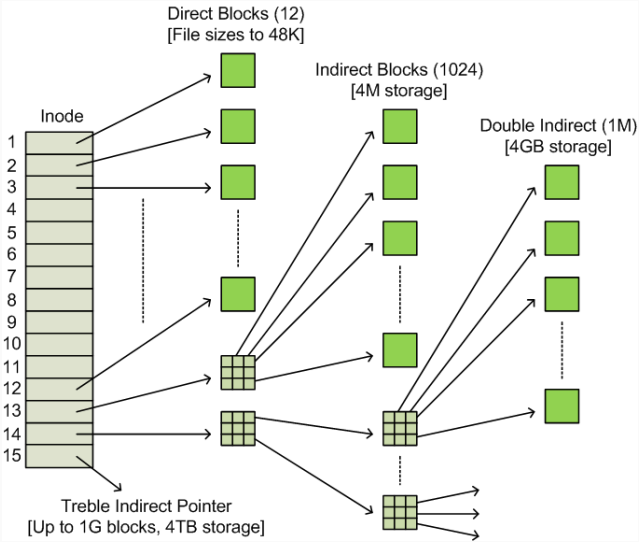
Blockgröße	1 KiB	2 KiB	4 KiB	64 KiB
Blöcke	2^{64}	2^{64}	2^{64}	2^{64}
Inodes	2^{32}	2^{32}	2^{32}	2^{32}
Dateisystemgröße	16 ZiB	32 ZiB	64 ZiB	1 YiB
Dateigröße (Extents)	4 TiB	8 TiB	16 TiB	256 TiB
Dateigröße (Blöcke)	16 GiB	256 GiB	4 TiB	256 PiB

Abbildung 5: ext4-Limits im 64-Bit-Modus [1]

- Standardgröße ist 4 KiB (und offizielles Maximum)
  - Sollte nicht größer als Seitengröße gewählt werden
- Unterschiedliche maximale Dateigrößen für Extents und Blöcke

1. Blockbasiert
   - Viele Blöcke gleicher Größe (üblicherweise 4 KiB)
   - Zeiger auf Blöcke im Inode
     - Direkt, indirekt, doppelt indirekt, dreifach indirekt
   - Hoher Overhead bei großen Dateien
     - Beispiel: 1 TiB große Datei benötigt 268.435.456 Zeiger
   - Beschränkt maximale Dateigröße

Allokation... [2]



2. Extentbasiert
   - Wenige möglichst große Extents
     - Vier Extents können im Inode gespeichert werden
     - Mehr in einer Baumstruktur und zusätzlichen Blöcken
   - Zeiger auf Startblock und Länge
     - Maximale Länge: 32.768 Blöcke
     - Entspricht 128 MiB bei einer Blockgröße von 4 KiB
   - Ermöglicht größere Dateien bei üblichen Blockgrößeren

- Blockallokation
  - Versuche zusammenhängende Blöcke zu allokieren
  - Versuche Blöcke in derselben Blockgruppe zu allokieren
- Multiblockallokation
  - Spekulativ 8 KiB bei Dateierzeugung allokieren
- Verzögerte Allokation
  - Blöcke werden erst allokiert, wenn auf das Speichergerät geschrieben werden muss

- Dateien und Verzeichnisse
  - Blöcke möglichst in der Blockgruppe des Inodes allokieren
  - Dateien möglichst in der Blockgruppe des Verzeichnisses allokieren
- Ziele der Allokationsstrategien
  - Möglichst große Zugriffe
    - Festplatten erreichen nur geringe IOPS-Werte
  - Zugriffe nahe beieinander
    - Reduziert Kopfbewegungen bei Festplatten
    - Metadaten der Blockgruppe eventuell schon im Cache
- Optimierungen bei SSDs weniger von Bedeutung

- Problem: Dateisystemoperationen benötigen mehrere Schritte
  - Z. B. das Löschen einer Datei
    1. Entfernen des Verzeichniseintrags
    2. Freigeben des Inodes
    3. Freigeben der Datenblöcke
  - Problematisch im Fall eines Absturzes
- Journaling zur Sicherung der Konsistenz des Dateisystems

- Geplante Änderungen werden ins Journal eingetragen
  - Entfernen wenn Operation vollständig durchgeführt
- Prüfung bei anschließender Dateisystemüberprüfung
  - Änderungen werden wiederholt oder verworfen
- Unterschiedliche Modi
  - Metadaten-Journaling und volles Journaling

- Journal: Alle Änderungen werden ins Journal geschrieben
  - Deaktiviert verzögerte Allokation und `O_DIRECT`
- Ordered: Metadaten werden ins Journal geschrieben
  - Zugehörige Daten werden vor Metadaten geschrieben
  - Problematisch mit verzögerter Allokation
  - Ist die Standardeinstellung
- Writeback: Metadaten werden ins Journal geschrieben
  - Bietet höchste Leistung aber geringste Sicherheit

Funktionen

- „Dateisystem Light“
  - Dünne Abstraktionsschicht über Speichergeräten
  - Objektbasierter Zugriff auf Daten
- Nur Grundoperationen
  - Erstellen, Öffnen, Schließen, Lesen, Schreiben
  - Manchmal nur Lesen bzw. Schreiben gesamter Objekte möglich
- Mögliche Unterstützung für Object Sets
  - Können benutzt werden um verwandte Objekte zu gruppieren

Funktionen...

- Üblicherweise keine Pfade
  - Zugriff über eindeutige IDs
  - Kein Overhead durch Pfadauflösung
  - Dadurch auch flacher Namensraum
- Block-/Extent-Allokation
  - Einer der komplexesten und leistungsrelevantesten Aspekte
- Auf unterschiedlichen Abstraktionsebenen verfügbar
  - Festplatte, Dateisysteme, Cloudspeicher

- Können als Unterbau für Dateisysteme genutzt werden
  - Erlaubt Konzentration auf Dateisystemfunktionalität
  - Speicherverwaltung durch separate Schicht
- Bei lokalen Dateisystemen häufig nicht sinnvoll
  - Funktionalität größtenteils durch POSIX vorgegeben
  - Hauptunterschied ist Blockallokation
- Sehr sinnvoll für parallele verteilte Dateisysteme
  - Kein redundanter Dateisystem-Overhead

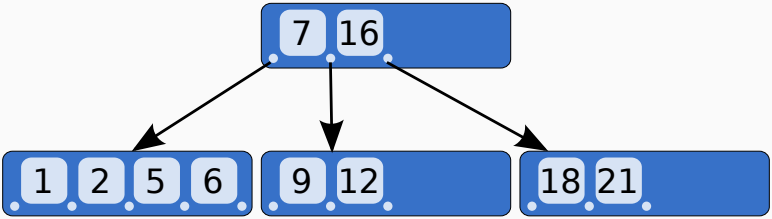


Abbildung 6: B-Baum [4]

- Verallgemeinerter Binärbaum
- Optimiert für Systeme, die große Blöcke lesen/schreiben
  - Zeiger und Daten gemischt

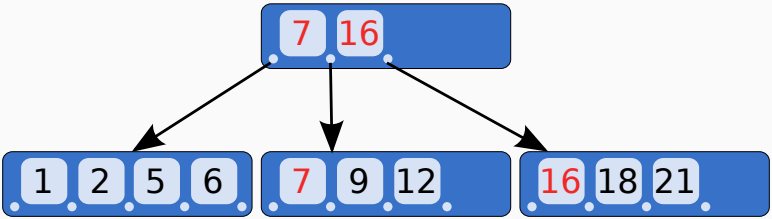


Abbildung 7: B+-Baum [4]

- Daten nur in Blättern
  - Vorteilhaft für Caching, da alle Knoten einfacher zu cachen sind
- Benutzt in NTFS, XFS etc.

Alternativen

- H-Baum
  - Basiert auf B-Baum
  - Andere Behandlung von Hash-Kollisionen
  - Benutzt in ext3 und ext4
- B^{ε}-Baum
  - Optimiert für Schreibvorgänge
  - Verbesserte Leistung für Einfügeoperationen, Bereichsabfragen und Aktualisierungen

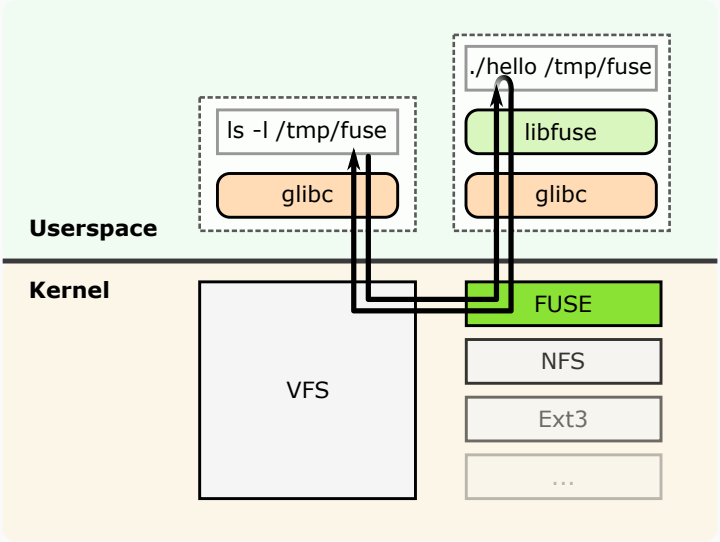
Leistungsbewertung

- Dateisystemleistung ist schwierig zu bewerten
  - Viele unterschiedliche Faktoren
  - Daten- vs. Metadatenleistung
  - Leistung unterschiedlicher Funktionen und Zugriffsmuster
  - Leistung für spezifische Anforderungen messen
- Datensicherheit kostet üblicherweise Leistung
  - Volles Journaling, Prüfsummen etc.

Kernel- vs. Userspace

- Dateisysteme üblicherweise direkt im Kernel implementiert
  - Hoher Wartungsaufwand
  - Komplexere Implementierung
- Alternative: Filesystem in Userspace (FUSE)
  - Besteht aus Kernelmodul und Bibliothek
  - Entwicklung von Dateisystemen als normale Prozesse
  - Umleitung in Userspace durch VFS und FUSE-Modul
  - Geringere Leistung durch Kontextwechsel

Kernel- vs. Userspace... [5]



Ausblick

- Lokale Dateisysteme sind häufig Basis für parallele verteilte Dateisysteme
  - Existierende und optimierte Blockallokation etc.
  - Object Stores wären häufig besser geeignet
- Moderne Dateisysteme integrieren zusätzliche Funktionen
  - Volumenverwaltung, Prüfsummen, Schnappschüsse etc.
  - Komfort und Datensicherheit zunehmend wichtig

Zusammenfassung

- Dateisysteme organisieren Daten und Metadaten
  - Üblicherweise standardisierte Schnittstelle
- Hauptobjekte sind Dateien und Verzeichnisse
  - Inodes speichern Metadaten
- Neue Techniken zur Effizienzsteigerung
  - Journaling um Konsistenz sicherzustellen
  - Speicherallokation mit Hilfe von Extents
  - Baumstrukturen für skalierbaren Zugriff

Quellen

[1] djwong. **Ext4 Disk Layout.**
https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[2] Hal Pomeranz. **Understanding Indirect Blocks in Unix File Systems.**
http://digital-forensics.sans.org/blog/2008/12/24/understanding-indirect-blocks-in-unix-file-systems.

[3] Werner Fischer and Georg Schönberger. **Linux Storage Stack Diagramm.**
https://www.thomas-krenn.com/de/wiki/Linux_Storage_Stack_Diagramm.

[4] Wikipedia. **B-tree.** http://en.wikipedia.org/wiki/B-tree.

[5] Wikipedia. **Filesystem in Userspace.**
http://en.wikipedia.org/wiki/Filesystem_in_Userspace.