TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

ROBUSTE DATEISYSTEME

CARSTEN WEINHOLD

GLIEDERUNG

- Dateisystemstrukturen
- Inkonsistenzen nach Abstürzen
- Konsistenzmechanismen:
  - Synchrones Schreiben
  - Soft Updates
  - Journaling
  - Log-strukturiert
  - Copy-on-write / Shadow Paging

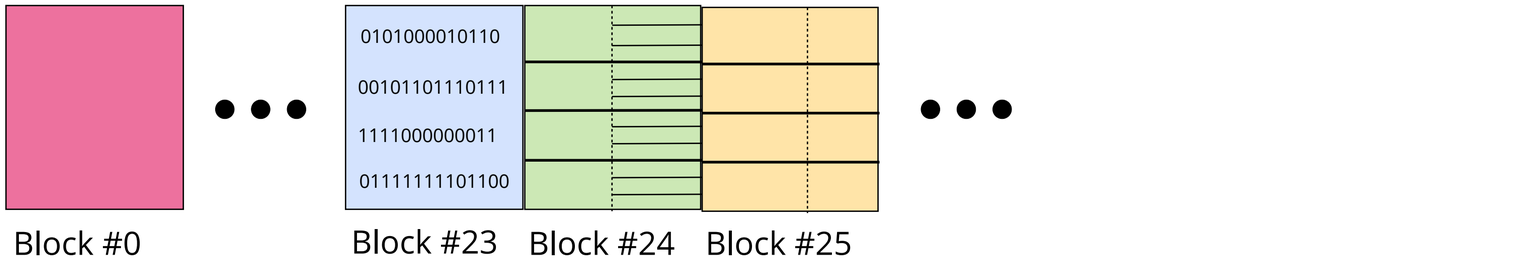
DATEISYSTEME

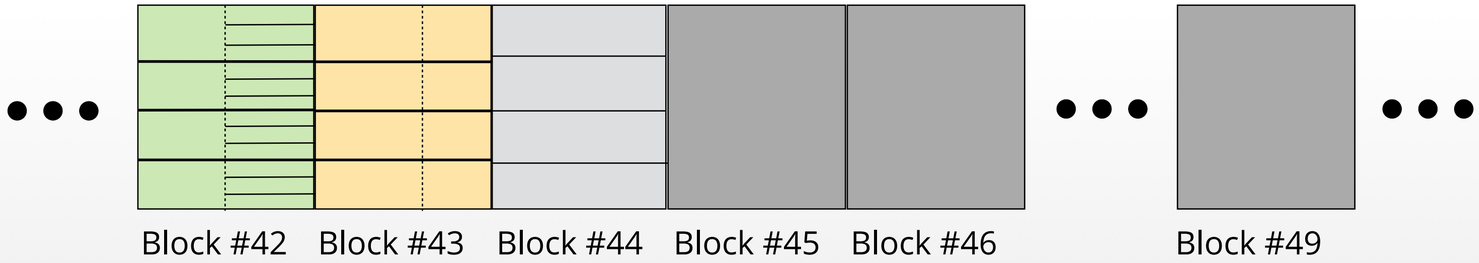
- Dateisystem: Abbildung von Objekten (Dateien) auf Speicherorte (Blöcke)
- Abbildung beschrieben durch Metadaten:
  - Hierarchie von Dateiverzeichnissen: Einträge bilden Dateinamen auf Inodes ab
  - Inodes mit Attributen und Zeigern
  - Zeiger benennen Blöcke mit Dateiinhalten
  - Status von Inodes, Blöcken: frei / belegt
  - Metadaten ebenfalls in Blöcken gespeichert

DATENSTRUKTUREN

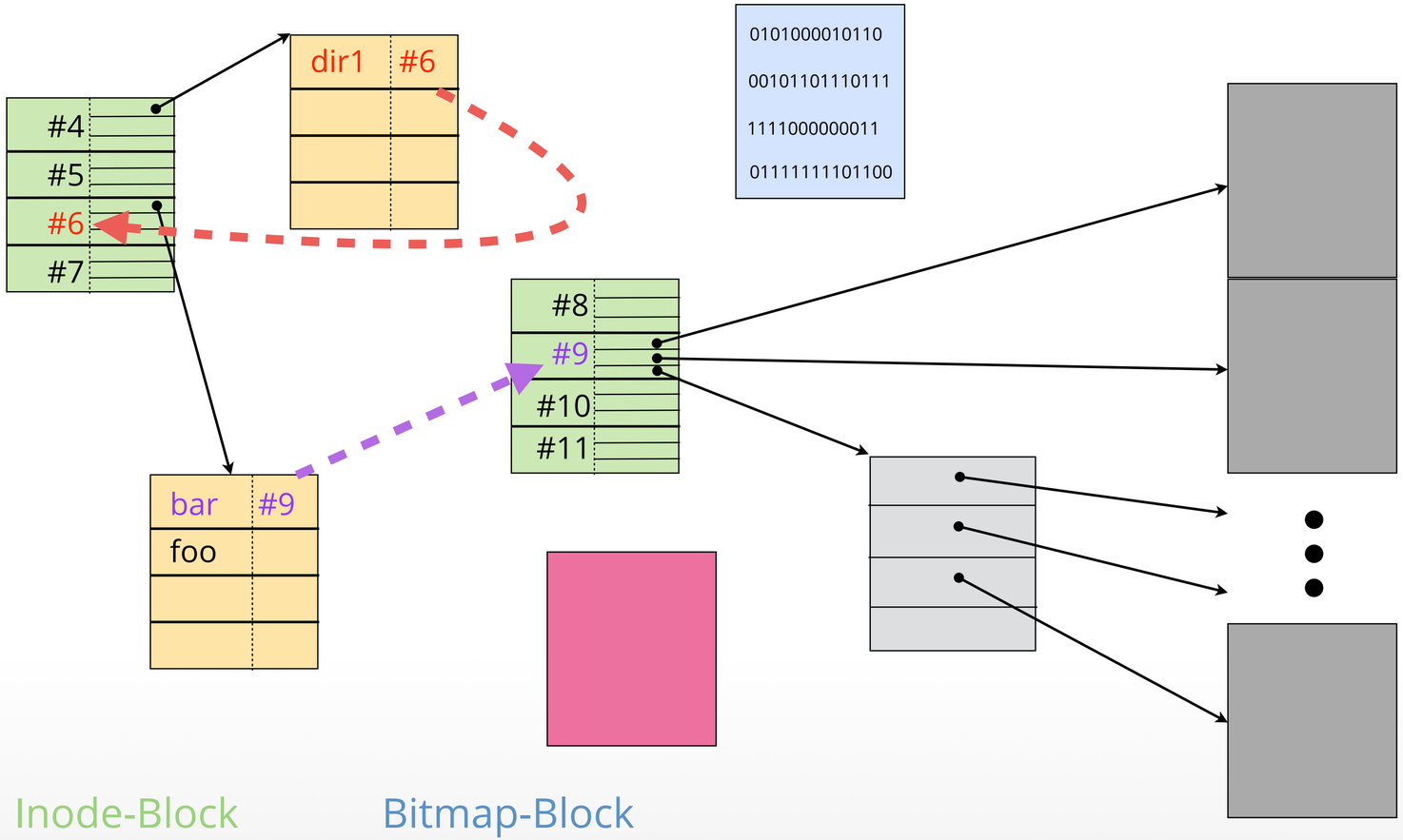
Blöcke auf Speichermedium linear durch Blocknummern adressiert

Lesen und Schreiben von Dateisystemstrukturen erfolgt blockweise





DATENSTRUKTUREN



Inode-Block
Verzeichnisblock
Indirect-Block
Bitmap-Block
Datenblock
Superblock

Dateisystemstrukturen für Datei mit Pfad „…/dir1/bar“

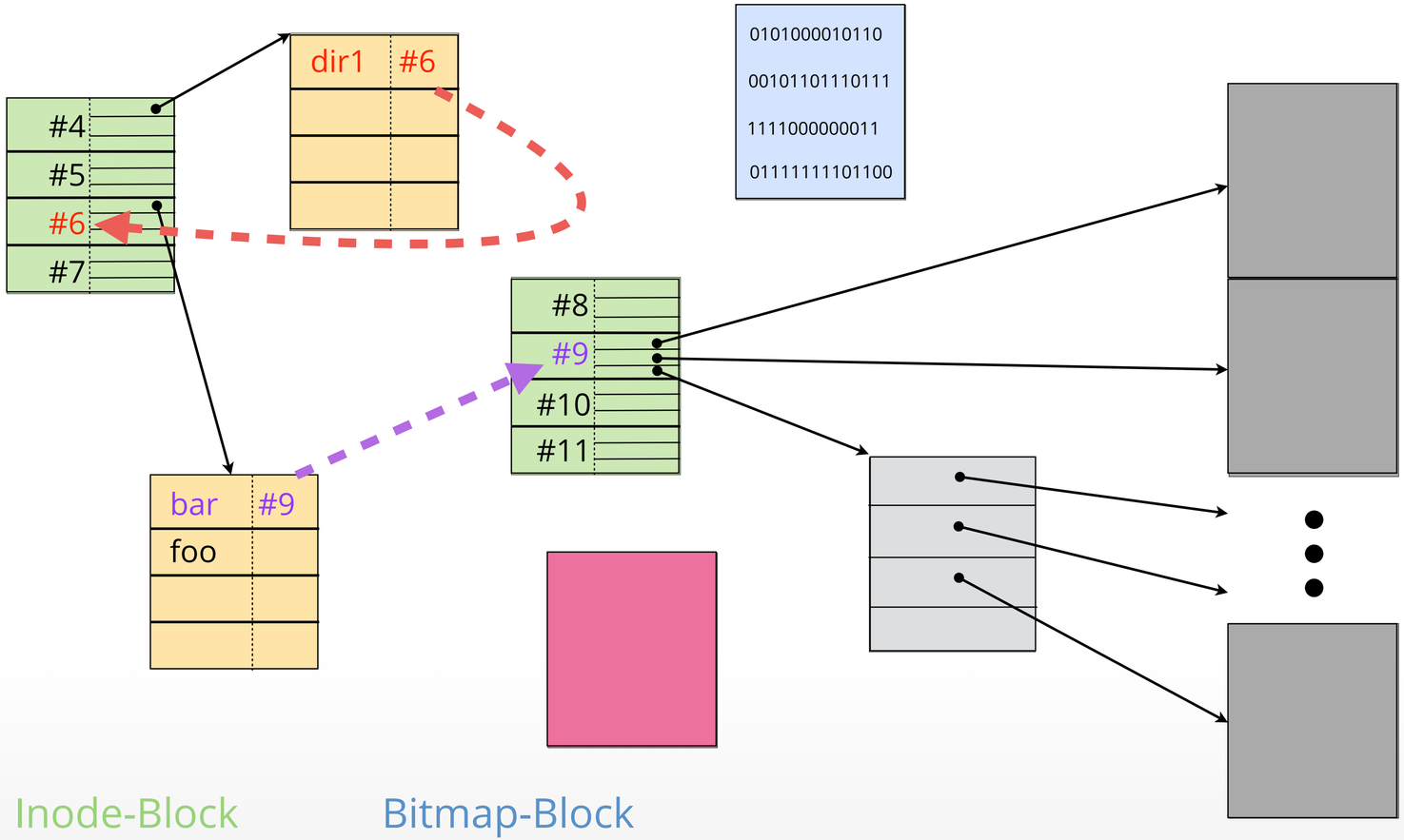
GLIEDERUNG

- Dateisystemstrukturen
- **Inkonsistenzen nach Abstürzen**
- Konsistenzmechanismen:
  - Synchrones Schreiben
  - Soft Updates
  - Journaling
  - Log-strukturiert
  - Copy-on-write / Shadow Paging

KONSISTENZPROBLEM

- Änderung an Dateisystem in Buffer Cache
- Abhängigkeiten zwischen Teiländerungen:
  - Metadaten und Dateiinhalte
  - Innerhalb von Metadaten
- Abhängigkeiten auch zwischen Blöcken:
  - Problem: Schreiben mehrerer Blöcke nicht atomar (oft nur einzelne Sektoren)
  - Unterbrechung (z.B. durch Stromausfall) beim Schreiben voneinander abhängiger Blöcke führt zu Inkonsistenz!

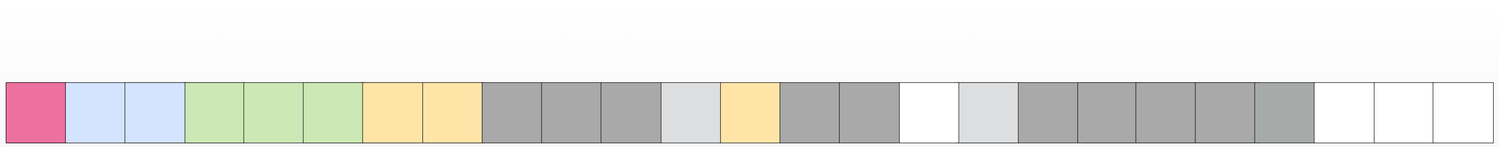
DATENSTRUKTUREN



Inode-Block
Verzeichnisblock
Indirect-Block
Bitmap-Block
Datenblock
Superblock

BUFFER CACHE

Jede Art von Zugriff (lesend/schreibend) auf Blockinhalte erfordert, dass zuvor einen Kopie des Blocks im Arbeitsspeicher angelegt wird. Dazu müssen Blöcke in der Regel zunächst vom Speichermedium eingelesen werden. Die vom Betriebssystem zentral verwalteten Blockkopien werden im **Buffer Cache** vorgehalten.

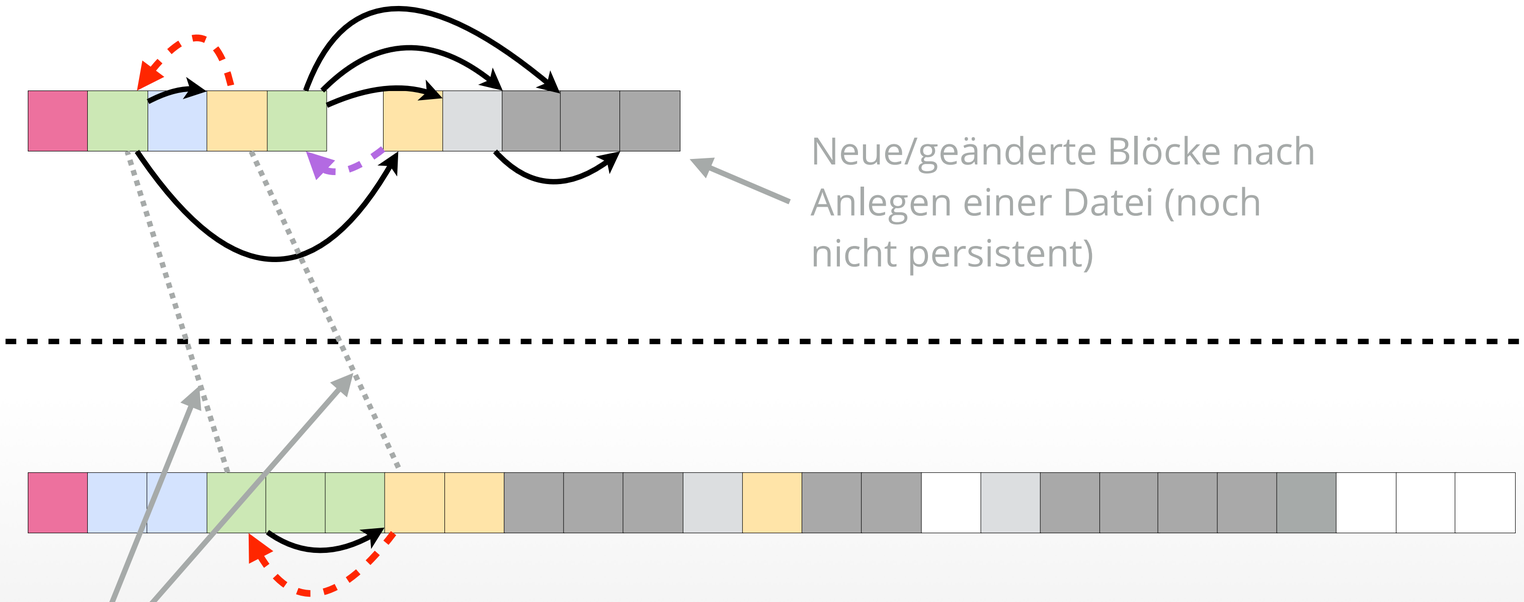


Dauerhafte (persistente) Version aller Blöcke auf Speichermedium (Festplatte, SSD, ...)

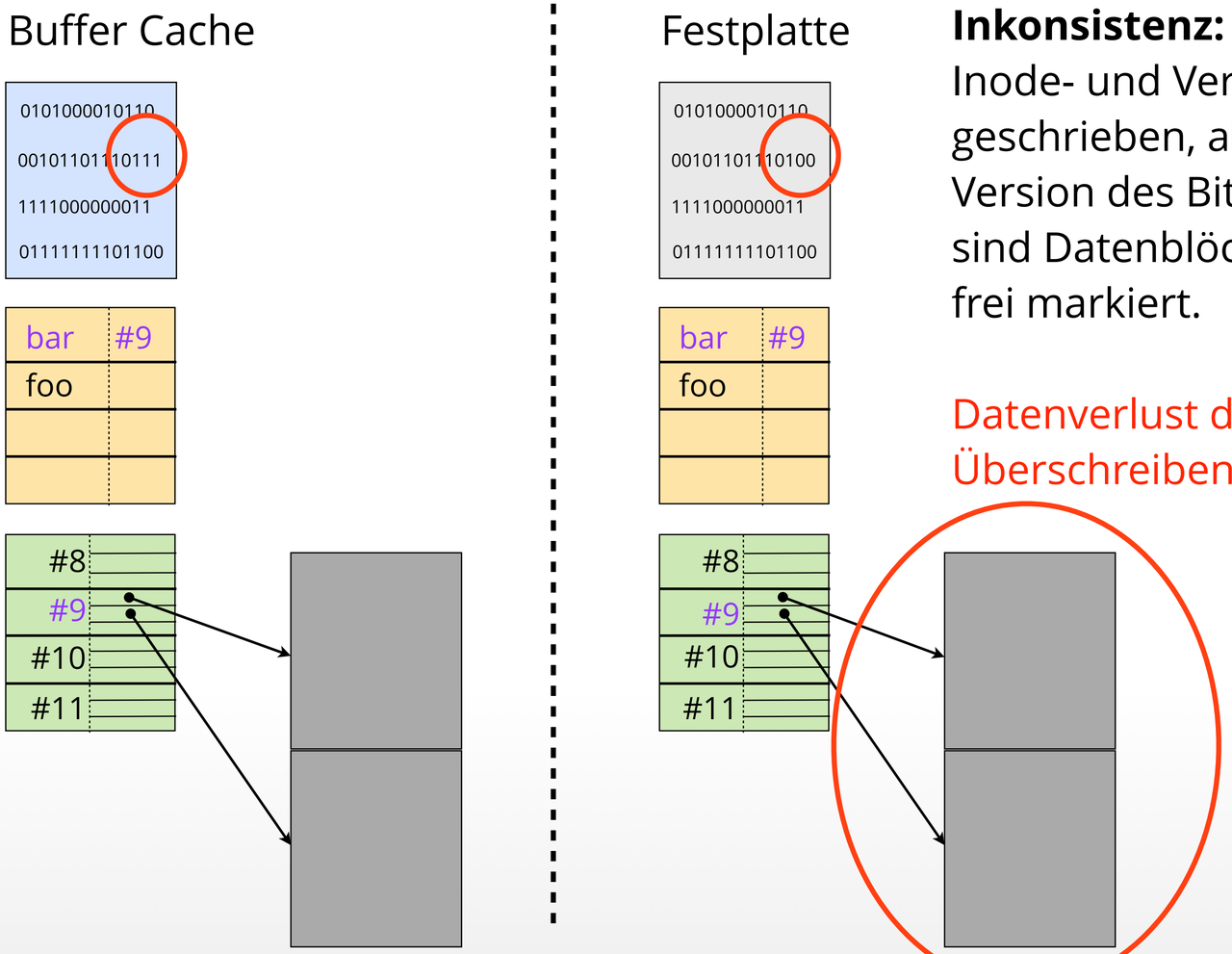
BUFFER CACHE

Alle Änderungen an Blöcken erfolgen zunächst im **Buffer Cache** und müssen anschließend auf das Speichermedium zurück geschrieben werden (**Sync**-Operation).

Wird nur eine Teilmenge der geänderten Blöcke im Buffer Cache zurückgeschrieben, können in den dauerhaften (persistenten) Blockkopien auf dem Speichermedium Inkonsistenzen auftreten!



BLOCK-BITMAP UNGÜLTIG



Inkonsistenz: Datenblöcke, Inode- und Verzeichnisblock geschrieben, aber in alter Version des Bitmap-Block sind Datenblöcke noch als frei markiert.

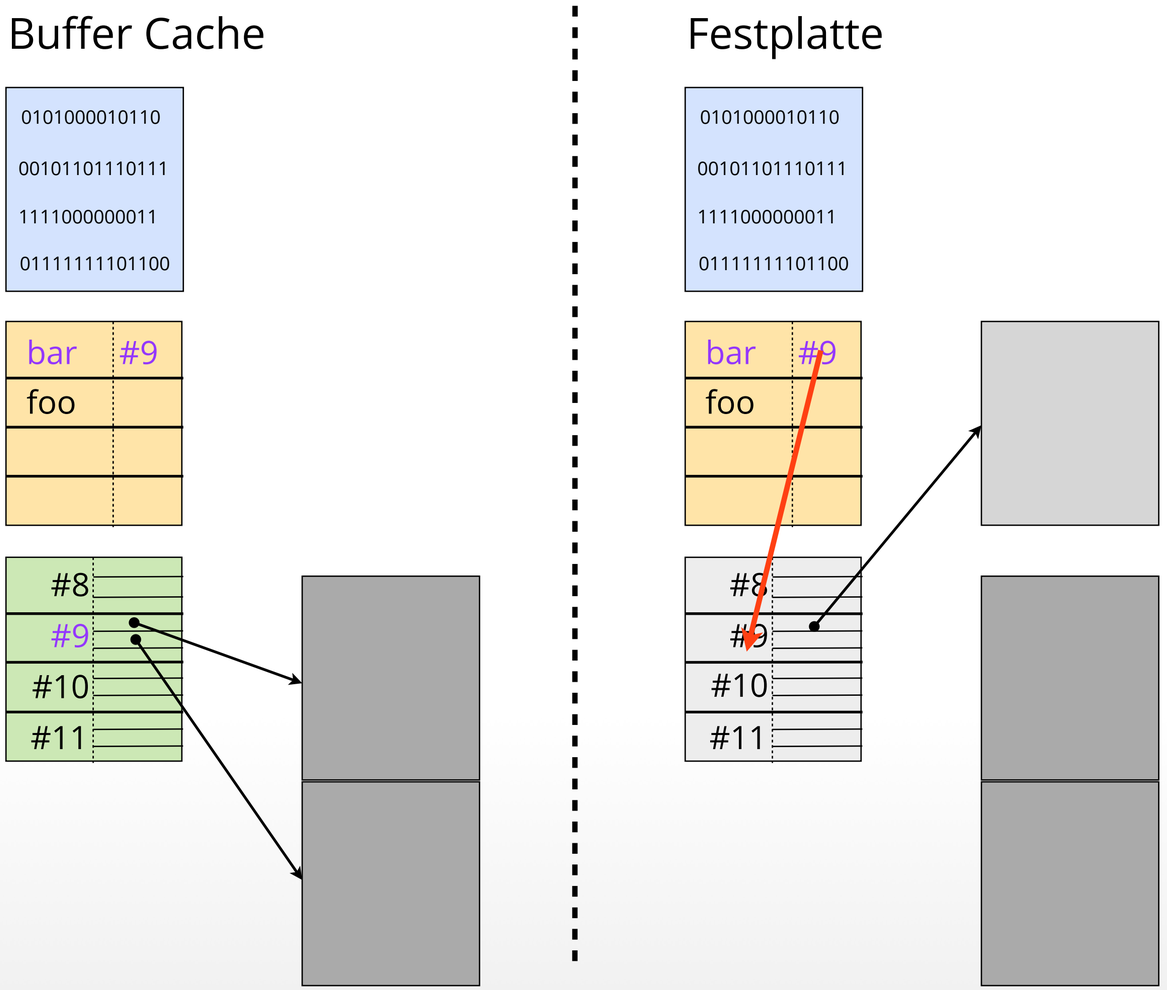
Datenverlust durch Überschreiben droht!

Absturz!

(vor Schreiben des Bitmap-Blocks)

Inode-Block / Verzeichnisblock / Indirect-Block / Bitmap-Block / Datenblock

UNGÜLTIGE ZEIGER



Inkonsistenz: Bitmap-, Verzeichnis- und Datenblöcke bereits geschrieben, aber Inode-Block noch nicht aktualisiert.

Absturz!

(vor Schreiben des Inode-Blocks)

Falsche/gelöschte Dateiinhalte werden referenziert!

Belegt- und Frei-Status von Blöcken inkorrekt.

Inode-Block / Verzeichnisblock / Indirect-Block / Bitmap-Block / Datenblock

INKONSISTENZEN

- **Kritisch:** (Verlust bereits persistenter Daten)
  - Zeiger auf falsche Inodes / Blöcke aus gelöschten / anderen Dateien
  - Belegter Block / Inode als frei markiert
  - Wert von Referenzzähler in Inode zu niedrig
- **Unkritisch:** (temporäre Ressourcenlecks)
  - Freier Block / Inode als belegt markiert
  - Referenzzähler in Inode zu hoch
  - Datenblock (oder Inode) geschrieben, aber Blockzeiger (oder Verzeichniseintrag) nicht

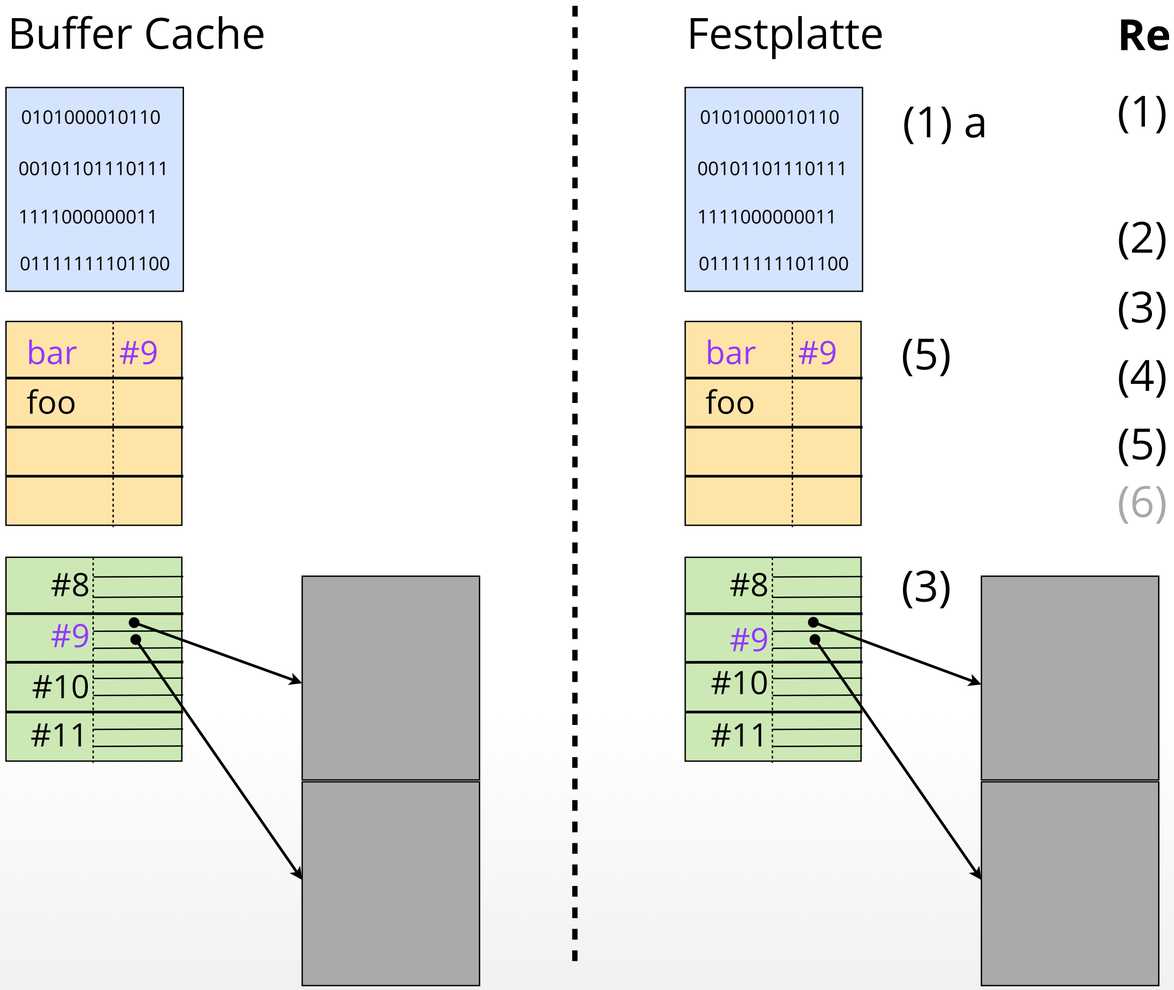
GLIEDERUNG

- Dateisystemstrukturen
- Inkonsistenzen nach Abstürzen
- Konsistenzmechanismen:
  - **Synchrones Schreiben**
  - Soft Updates
  - Journaling
  - Log-strukturiert
  - Copy-on-write / Shadow Paging

SYNCHRONES SCHREIBEN

- **Idee:** Modifizierte Blöcke sofort in sicherer Reihenfolge schreiben:
  - Neue Blöcke: zunächst Allokation in Bitmap-Block schreiben, dann neuen Block
  - Erst Block schreiben, dann Zeiger darauf
- Schreiben von Blöcken erfolgt synchron:
  - Metadatenblöcke: Warten auf Rückmeldung von Speichergerät **(Write Barrier)**
  - Datenblöcke: Warten bei letztem Block reicht

SICHERE REIHENFOLGE



Reihenfolge der Operationen:

(1) Bitmap- und Datenblöcke (a-c) schreiben
(2) [Write Barrier]
(3) Inode-Block schreiben
(4) [Write Barrier]
(5) Verzeichnisblock schreiben
(6) [Write Barrier] (wichtig für nachfolgende Operationen)

Inode-Block / Verzeichnisblock / Indirect-Block / Bitmap-Block / Datenblock

SYNCHRONES SCHREIBEN

Beispiel: neue (leere) Datei anlegen

1) Inode allokieren, Inode-Bitmap schreiben — 2) [Write Barrier]

3) Inode initialisieren, Inode-Block schreiben — 4) [Write Barrier]

5) Verzeichniseintrag schreiben:

a) Bei Bedarf: neuen Verzeichnisblock allokieren, Block-Bitmap schreiben, inkl. zusätzlicher Write Barriers (hier nicht diskutiert)

b) Neuen/modifizierten Verzeichnisblock schreiben — 6) [Write Barrier]

c) Inode für Verzeichnis aktualisieren (Zeitstempel, Größe), Inode-Block schreiben — 7) [Write Barrier]

(nur wichtig für nachfolgende Operationen)

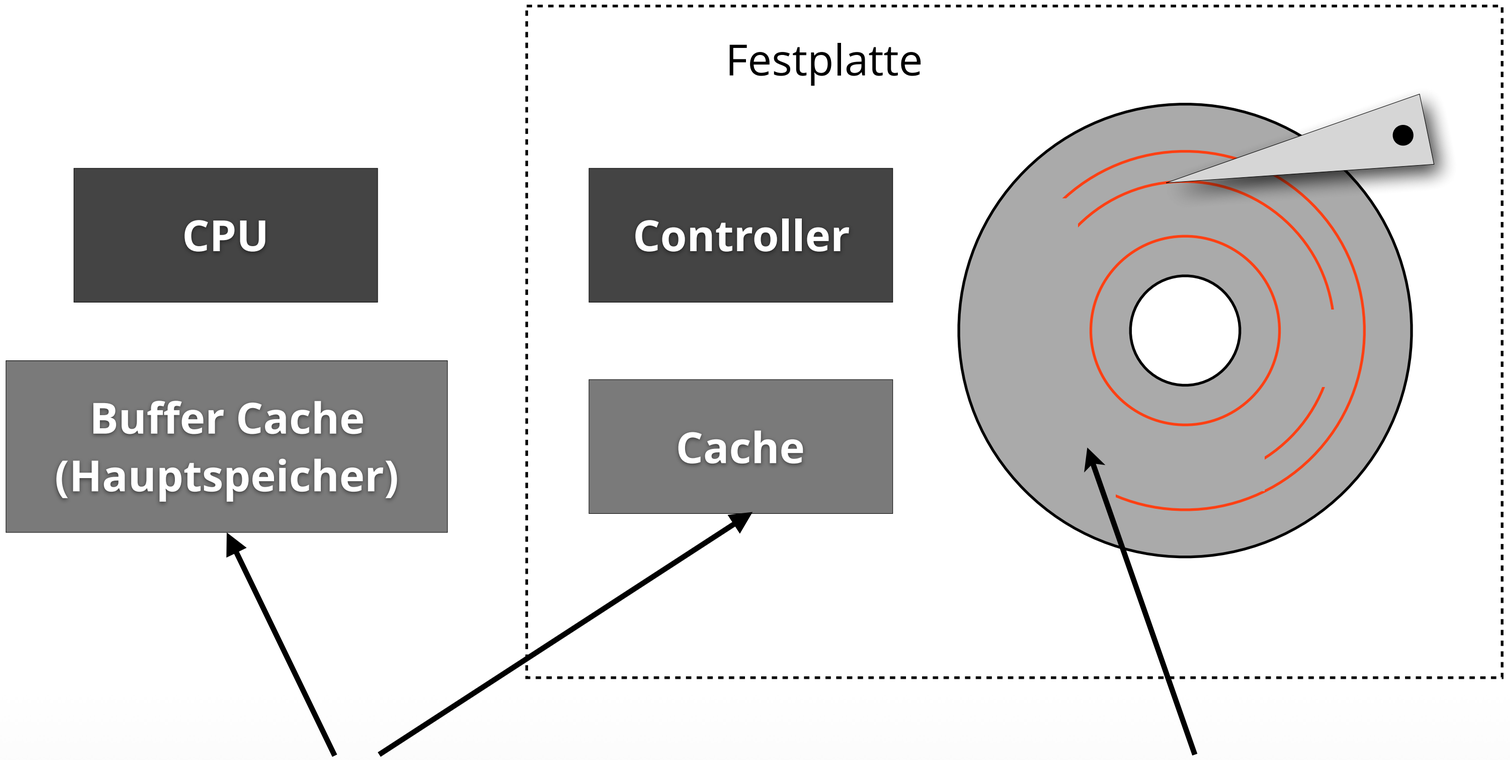
KONSISTENZREGELN

- Grundlegende Regeln für Konsistenz:
  - **Gültige Zeiger:** Setze keinen Zeiger auf eine Struktur, bevor diese initialisiert wurde
  - **Wiederverwendung:** Verwende keine Ressource erneut, bevor alle vorherigen Zeiger darauf invalidiert wurden
  - **Erreichbarkeit:** Invalidiere niemals den alten Zeiger auf eine gültige Ressource, bevor der neue Zeiger gesetzt wurde

EIGENSCHAFTEN

- **Konsistenz nach Absturz:**
  - Keine Zeiger auf nicht initialisierte Metadaten
  - Ressourcen-Lecks möglich: Inodes, Blöcke
  - Zu hohe Werte in Referenzzählern
  - **CLEAN**-Flag in Superblock nicht gesetzt
- **Korrektur:** Dateisystem-Check ***(fsck)***
- **Schlechte Performance:**
  - Nach Absturz: (sehr) zeitaufwändiger fsck-Lauf
  - Pro Operation: Mehrere Write Barriers (teuer!!!)

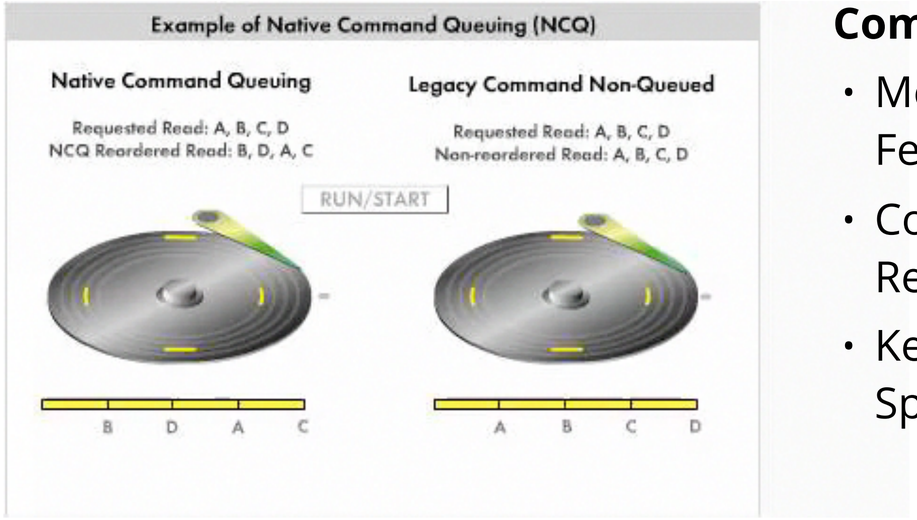
HARDWARE-REALITÄTEN



Caches: Betriebssystem verwaltet Buffer-Cache, Speichergerät hat eigenen Cache (oft ganze Tracks bei Festplatten)

Speichermedium: Daten- und Metadatenblöcke sind verteilt (Festplatte: Schreib-/Lesekopf muss Sektoren ansteuern)

HARDWARE-REALITÄTEN



Quelle: [1]

Command-Queuing:

- Mehrere Lese-/Schreib-aufträge können an Festplatte gesendet werden
- Controller entscheidet selbst über optimale Reihenfolge
- Keine Garantie für Reihenfolge persistenter Speicherung (aber Benachrichtigung)

Problematische Sicherstellung garantierter Schreibreihenfolge:

- Ältere Spezifikationen wie Serial-ATA kennen keine Write Barriers
  → Speichergerät darf ausstehende Schreiboperationen beliebig umsortieren
- Garantierte Persistenz eines Blocks vor einem anderen nur nach explizitem Zurückschreiben des internen Caches (FLUSH-Kommando)
- Vorteile von Command-Queuing gehen verloren

Neue Schnittstellen wie NVMe beheben dieses Problem

GLIEDERUNG

- Dateisystemstrukturen
- Inkonsistenzen nach Abstürzen
- Konsistenzmechanismen:
  - Synchrones Schreiben
  - **Soft Updates**
  - Journaling
  - Log-strukturiert
  - Copy-on-write / Shadow Paging

SOFT UPDATES

- **Idee:** Puffern, gemeinsames Schreiben
  - Viele Änderungen zunächst im Buffer Cache
  - Bei Zurückschreiben der Änderungen alle Abhängigkeiten zwischen Blöcken beachten
- **Umsetzung:**
  - Dependency-Strukturen für jeden Block
    - Metadaten (z.B.: Verzeichniseintrag -> Inode)
    - Daten (Inode / Indirect-Block -> Datenblock)
  - Problem: Zyklen in Blockabhängigkeiten!

ABHÄNGIGKEITSZYKLEN

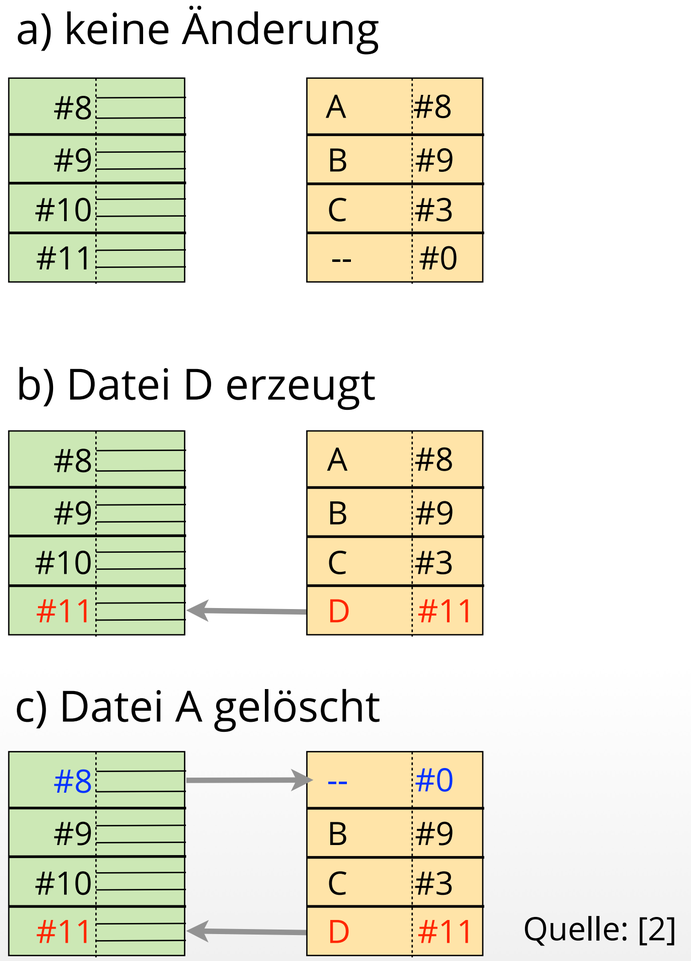
- Inode- und Verzeichnisblöcke im Buffer Cache:

  a) keine Schreibabhängigkeiten

  b) Inode-Block muss vor Verzeichniseintrag initialisiert sein

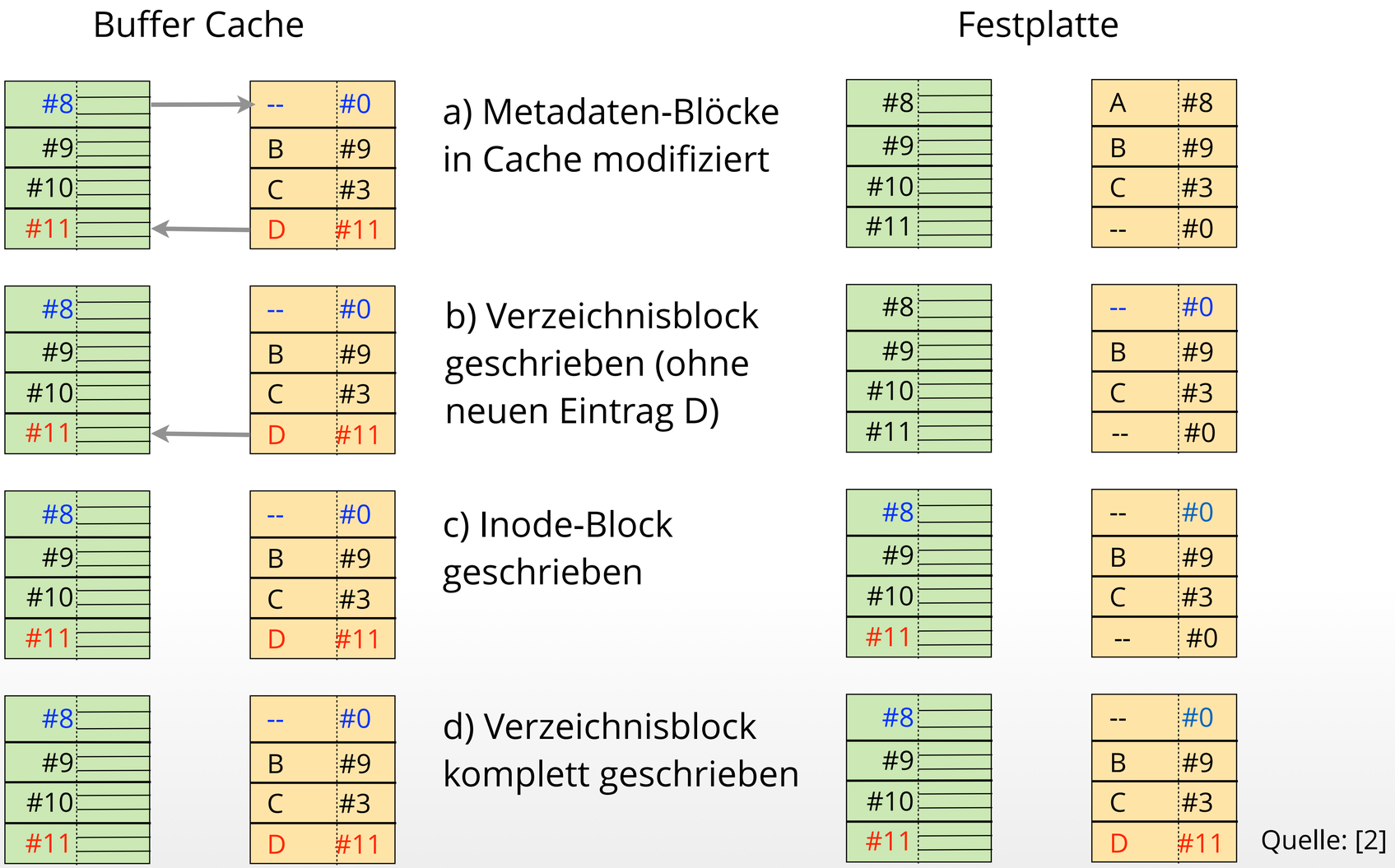
  c) Inode-Zeiger in Verzeichniseintrag muss vor Inode gelöscht werden

- Zyklische Abhängigkeit!



Quelle: [2]

ZYKLEN AUFBRECHEN



Quelle: [2]

ABHÄNGIGKEITEN

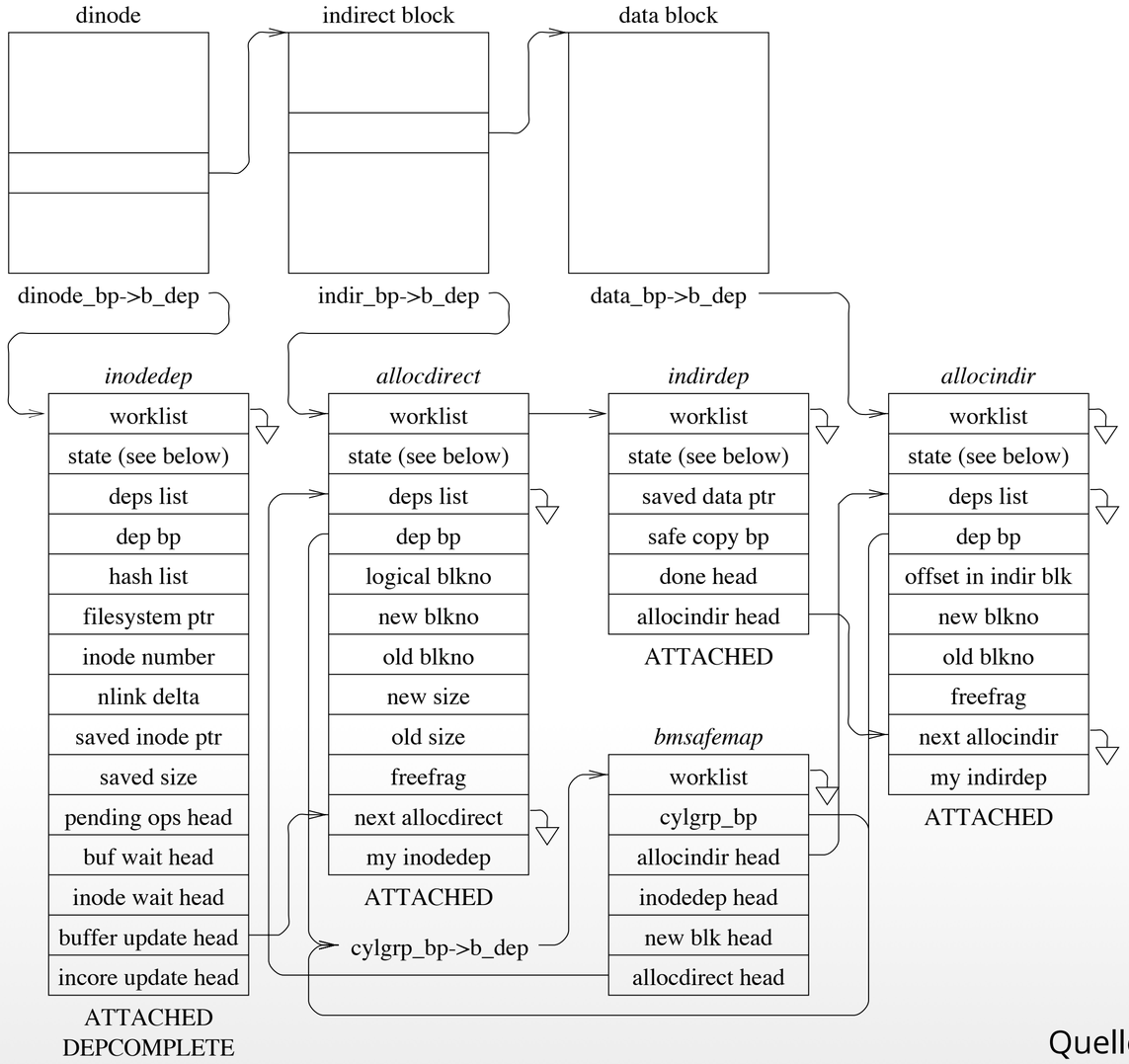
- Zurückschreiben von Blöcken jederzeit
- Soft-Updates-Code inspiziert Dependency-Strukturen vor Schreiboperation:
  - Konsistente Änderungen können persistent gemacht werden und werden übernommen
  - Änderungen mit nicht erfüllten Abhängigkeiten werden temporär zurückgerollt
  - Nach Schreiben der konsistenten Version des Blocks werden Änderungen wiederholt

ABHÄNGIGKEITEN

Name	Function	Associated Structures
bmsafemap	track bitmap dependencies (points to lists of dependency structures for recently allocated blocks and inodes)	cylinder group block
inodedep	track inode dependencies (information and list head pointers for all inode-related dependencies, including changes to the link count, the block pointers, and the file size)	inode block
allocdirect	track inode-referenced block (linked into lists pointed to by an inodedep and a bmsafemap to track inode's dependence on the block and bitmap being written to disk)	data block or indirect block or directory block
indirdep	track indirect block dependencies (points to list of dependency structures for recently-allocated blocks with pointers in the indirect block)	indirect block
allocindir	track indirect block-referenced block (linked into lists pointed to by an indirdep and a bmsafemap to track the indirect block's dependence on that block and bitmap being written to disk)	data block or indirect block or directory block
pagedep	track directory block dependencies (points to lists of diradd and dirrem structures)	directory block
diradd	track dependency between a new directory entry and the referenced inode	inodedep and directory block
mkdir	track new directory creation (used in addition to standard diradd structure when doing a mkdir)	inodedep and directory block
dirrem	track dependency between a deleted directory entry and the unlinked inode	first pagedep then tasklist
freefrag	tracks a single block or fragment to be freed as soon as the corresponding block (containing the inode with the now-replaced pointer to it) is written to disk	first inodedep then tasklist
freeblks	tracks all the block pointers to be freed as soon as the corresponding block (containing the inode with the now-zeroed pointers to them) is written to disk	first inodedep then tasklist
freefile	tracks the inode that should be freed as soon as the corresponding block (containing the inode block with the now-reset inode) is written to disk	first inodedep then tasklist

Quelle: [2]

ABHÄNGIGKEITEN

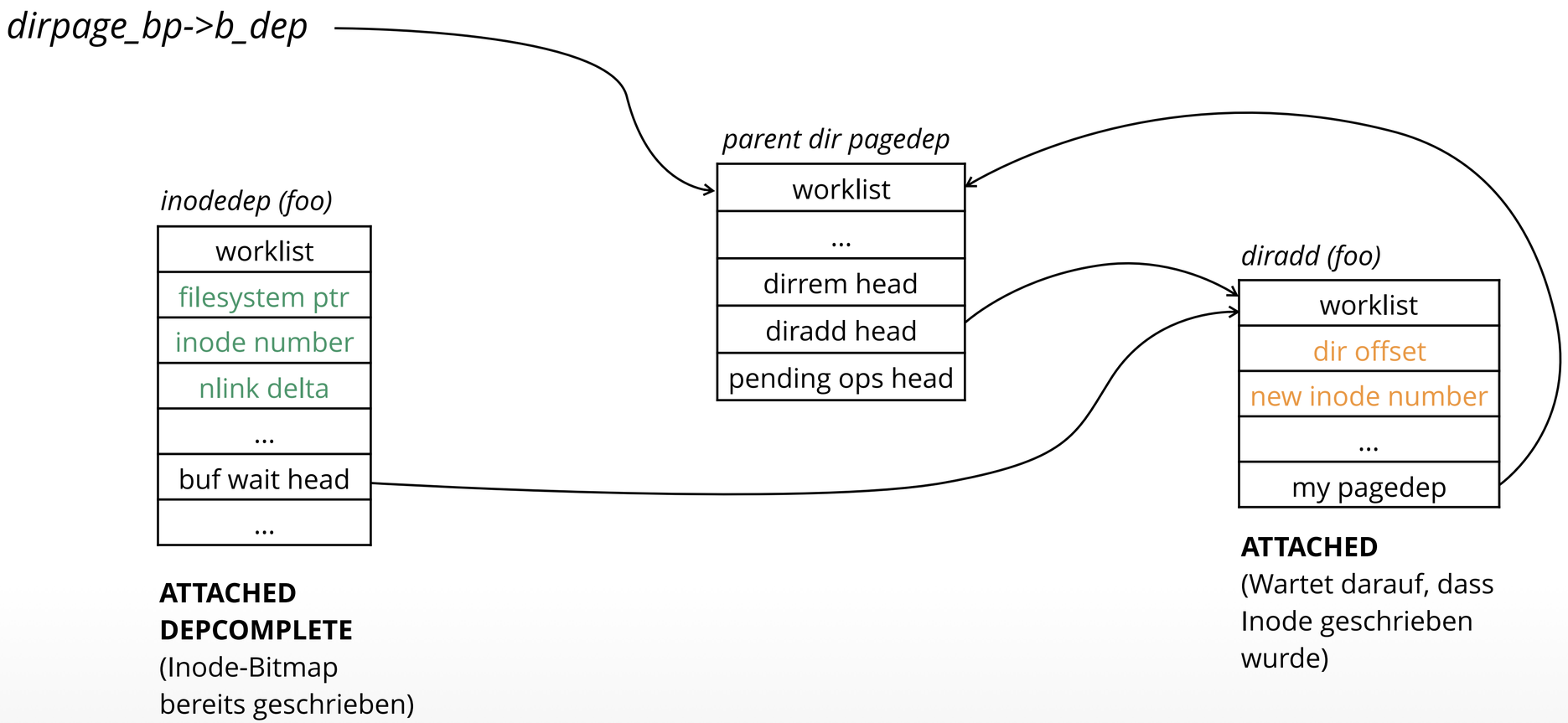


Quelle: [2]

ZUSTANDS-FLAGS

- **ATTACHED:**
  - Puffer wird im Moment nicht geschrieben
  - Bei Zurückrollen: ATTACHED-Flag wird gelöscht, nach Schreiben des Blocks werden Änderungen im Block wiederholt und Flag neu gesetzt
- **DEPCOMPLETE:**
  - Änderung sicher, kein Zurückrollen notwendig
- **COMPLETE:**
  - Änderung wurde auf Platte geschrieben
- Struktur freigeben? Nur wenn alle Flags gesetzt

ABHÄNGIGKEITEN



Beispiel: Abhängigkeiten für Anlegen eines neuen Verzeichniseintrags (vereinfachte Darstellung aus [2])

EIGENSCHAFTEN

- **Nach Absturz:** unkritische Inkonsistenzen
  - Freie Inodes / Blöcke als belegt markiert
  - Wert des Referenzzählers in Inodes zu hoch
  - Alter und neuer Dateiname, falls rename-Operation unterbrochen
- **Korrektur:**
  - fsck-Lauf zur Ressoucenfreigabe
  - FFS: fsck im Hintergrund, wenn eingehängt

EIGENSCHAFTEN

- **Gute Performance:**
  - Nur wenige Write Barriers
  - fsck-Lauf im Hintergrund
- **Hohe Komplexität:**
  - Tief verankert in Implementierung, stark abhängig von Dateisystem-Layout
  - Aufwändiges Verfolgen von Abhängigkeiten
  - Rollback und Rollforward von Änderungen

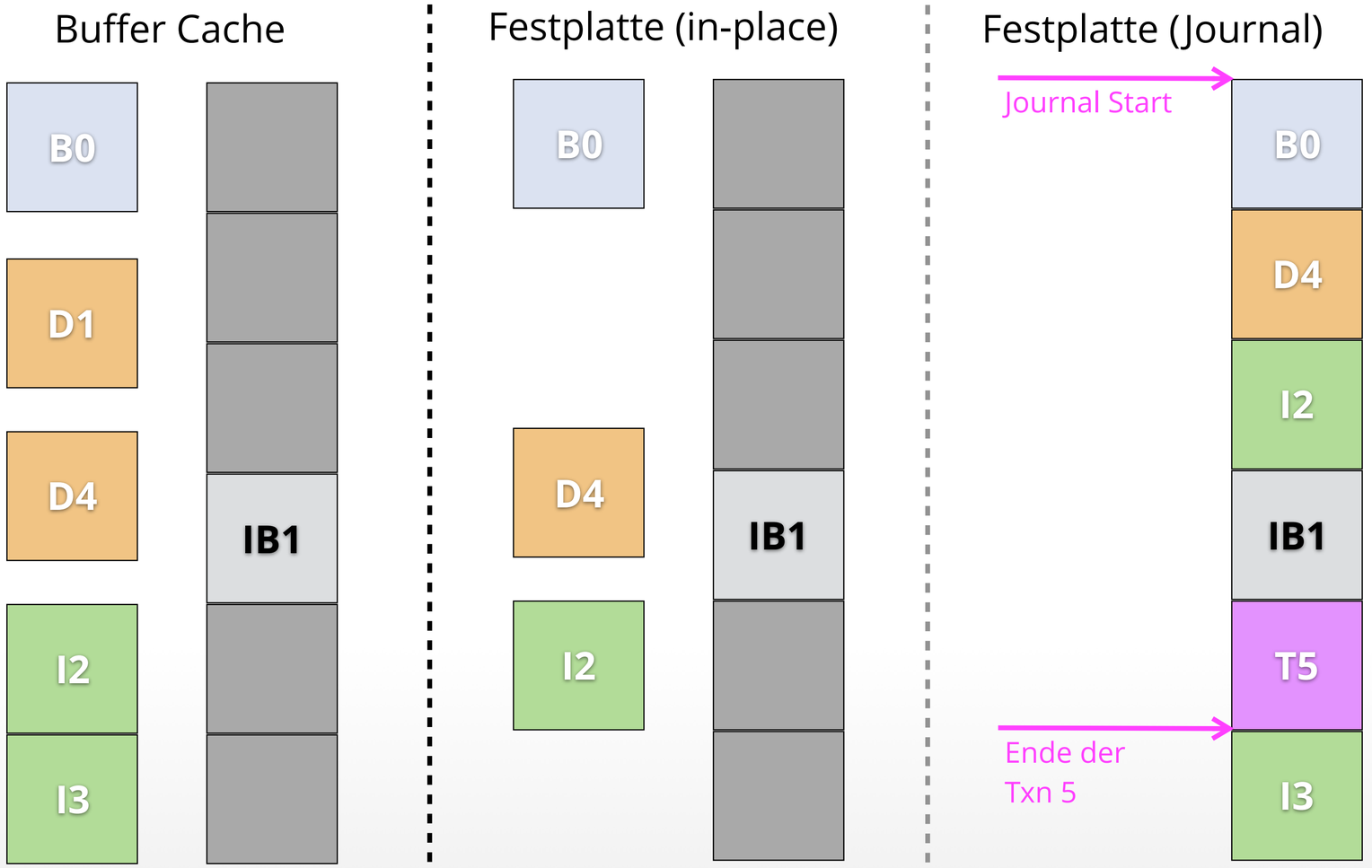
GLIEDERUNG

- Dateisystemstrukturen
- Inkonsistenzen nach Abstürzen
- Konsistenzmechanismen:
  - Synchrones Schreiben
  - Soft Updates
  - **Journaling**
  - Log-strukturiert
  - Copy-on-write / Shadow Paging

JOURNALING

- **Idee:** Write-ahead Log (Journal)
  1) Protokollierung geplanter Änderungen an Dateisystem-strukturen in **Journal**
  2) Journal-**Transaktion** als komplett markieren
  3) Änderungen „in-place“ in verteilte Dateisystemblöcke schreiben **(Checkpointing)**
  4) Transaktion in Journal freigeben
- **Nach Absturz:** in Journal protokollierte Transaktionen erneut „in-place“ schreiben

METADATEN-JOURNALING



Inode-Block / Verzeichnisblock / Indirect-Block / Bitmap-Block / Datenblock

Ablauf beim Zurückschreiben aus dem Buffer Cache

Beispiel-Transaktion Txn 5

Write-Back und Journaling-Phase:

1) Datenblöcke an Zielpositionen schreiben (in-place, d.h. bleiben dort)
2) Metadatenblöcke B0, D4, I2, IB1 hintereinander ins Journal schreiben
3) [Write Barrier]
4) Transaktionsblock T5 schreiben um Txn 5 als gültig zu markieren
5) [Write Barrier]

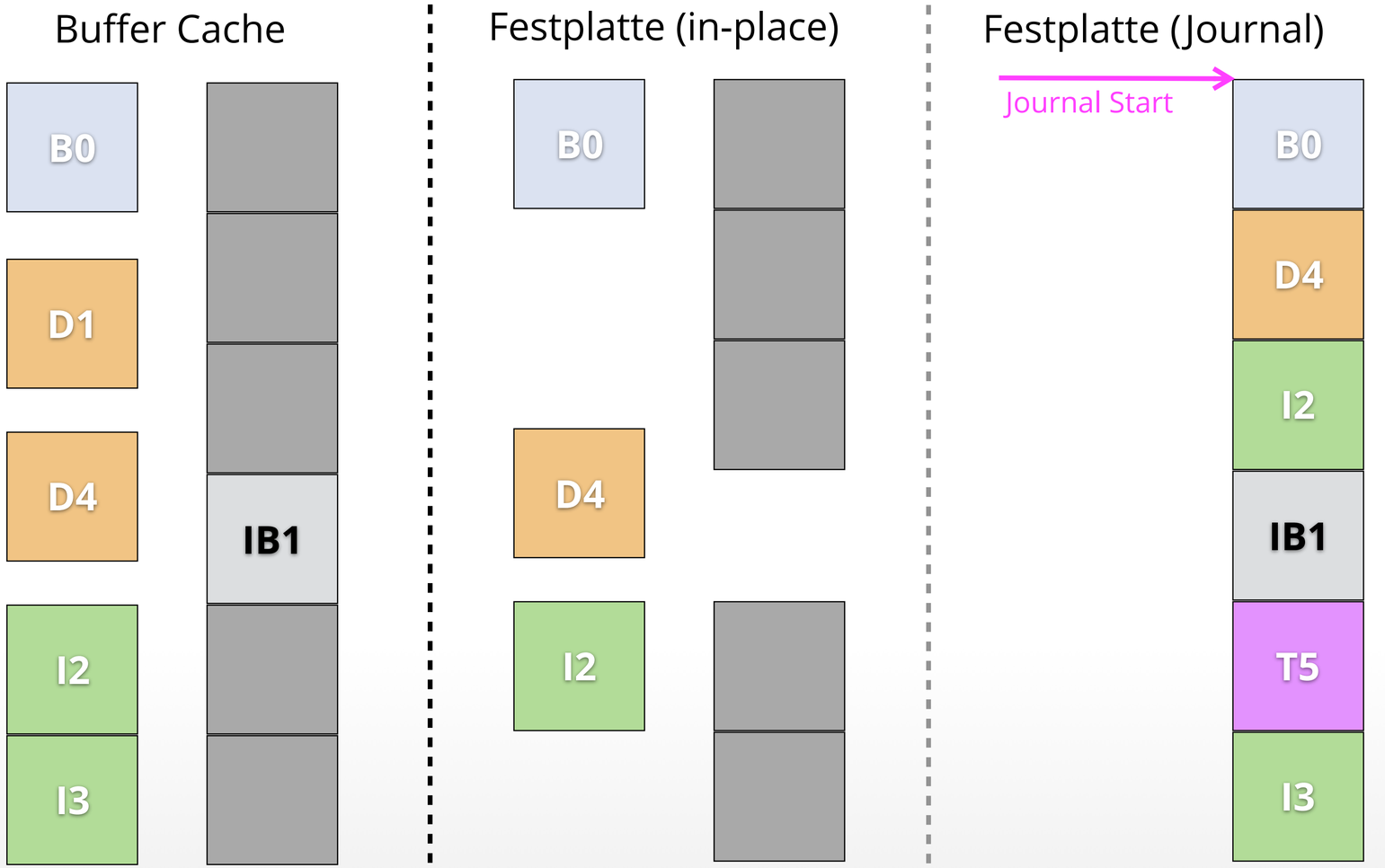
Checkpointing-Phase:

6) Metadatenblöcke B0, D4, I2, IB1 an Zielpositionen (in-place) schreiben
7) [Write Barrier]

Txn 5 abgeschlossen

8) Txn 5 aus Journal löschen (Anfang wird verschoben auf Position hinter Txn 5)

JOURNALING: ABSTURZ



Absturz
(während Checkpointing-Phase)

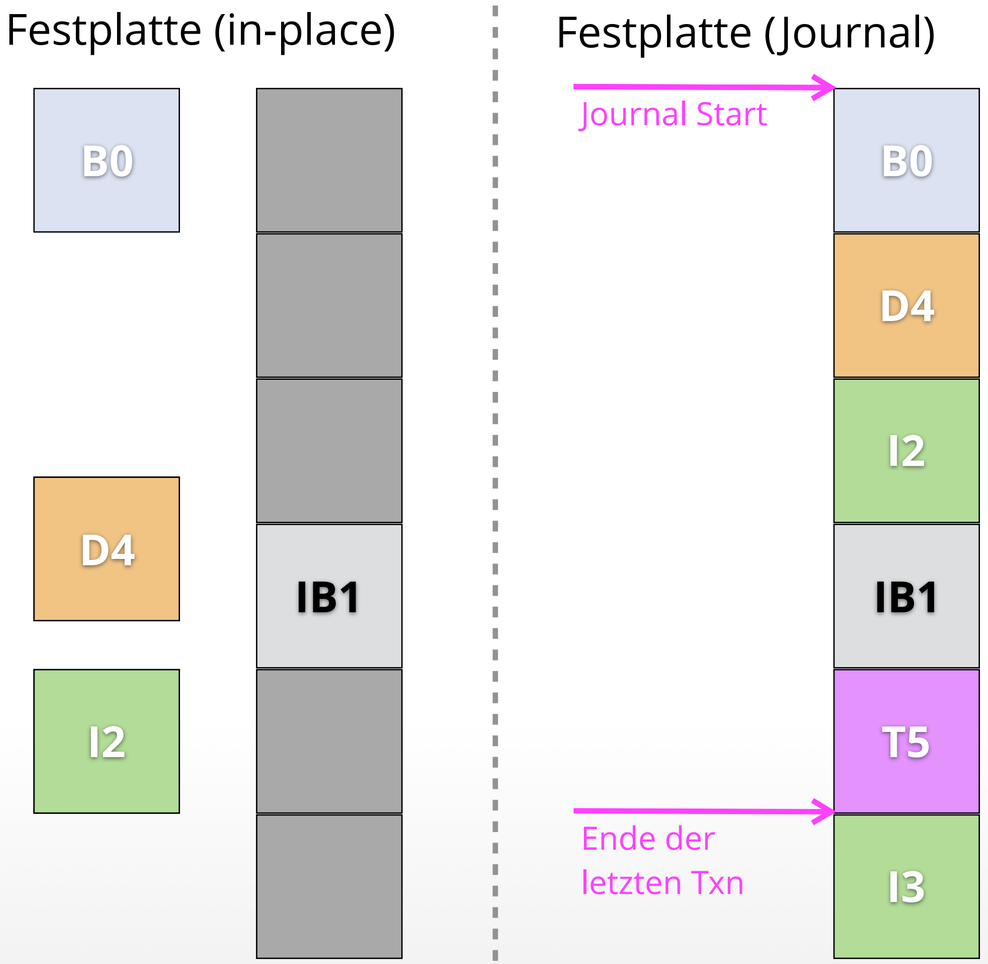
Inode-Block / Verzeichnisblock / Indirect-Block / Bitmap-Block / Datenblock

TRANSAKTIONS-REPLAY

Fall 1: Absturz während Checkpointing

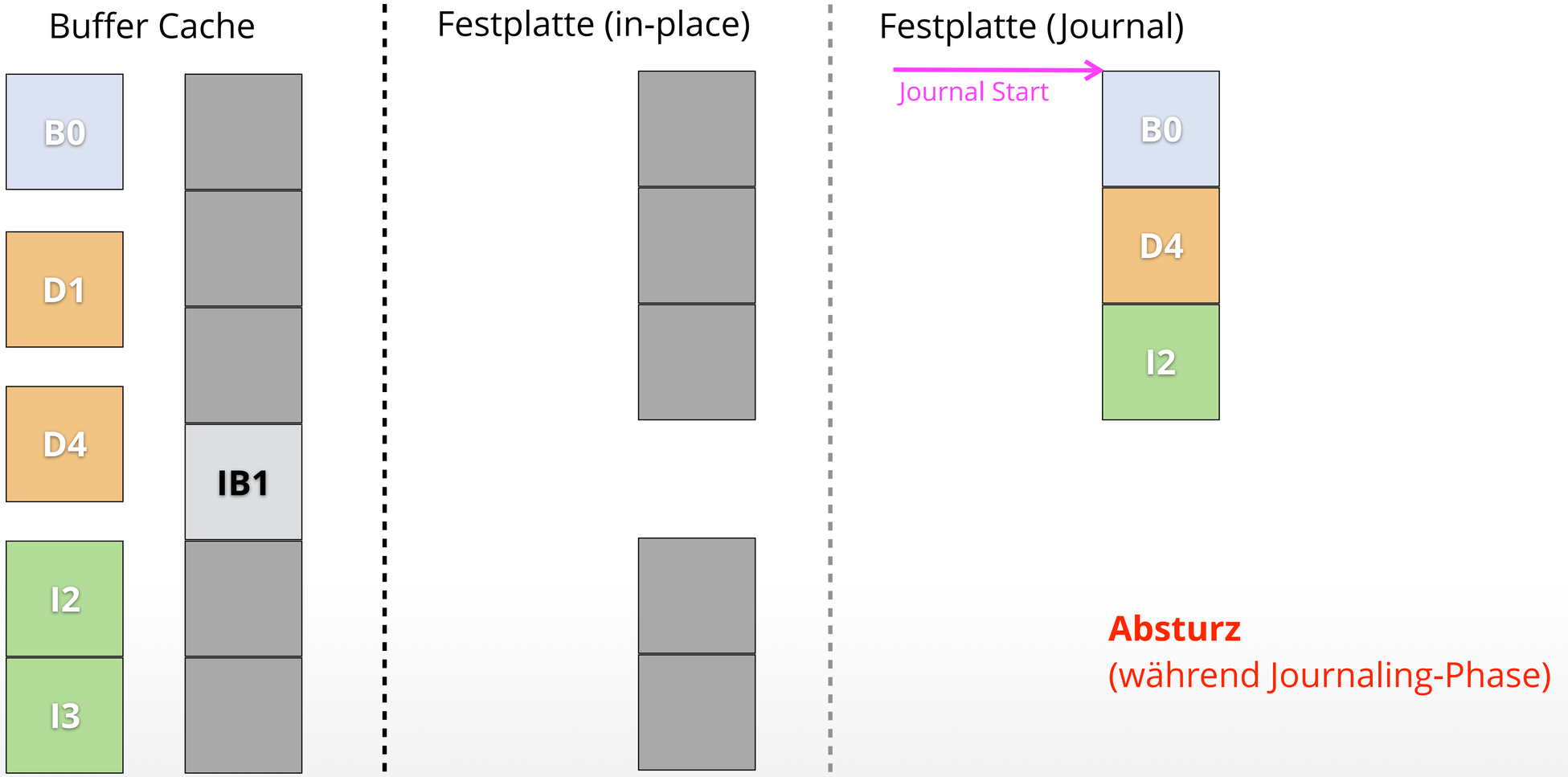
Datenblöcke geschrieben, alle Metadatenblöcke in Journal

Komplette Transaktion wird wiederholt, Metadaten an in-place-Adressen für neue Datei vervollständigt!



Inode-Block / Verzeichnisblock / Indirect-Block / Bitmap-Block / Datenblock

JOURNALING: ABSTURZ



Absturz
(während Journaling-Phase)

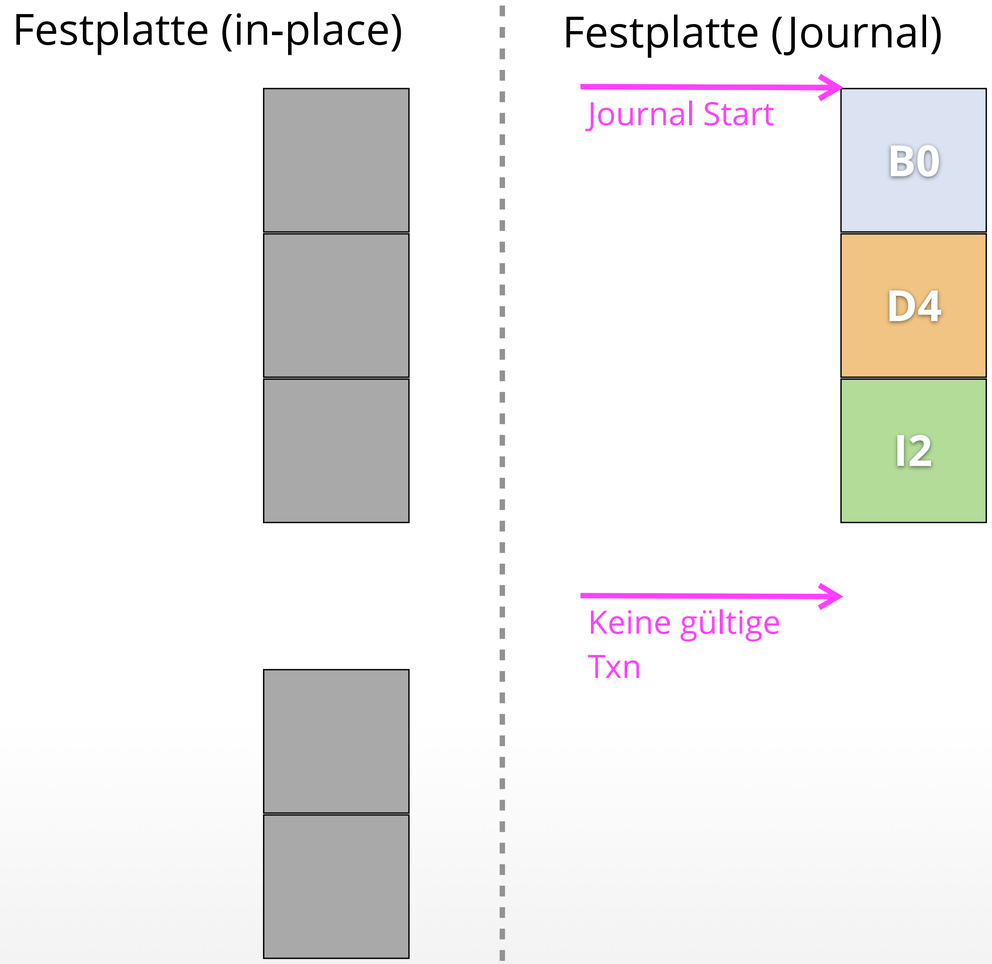
Inode-Block / Verzeichnisblock / Indirect-Block / Bitmap-Block / Datenblock

TRANSAKTIONS-REPLAY

Fall 2: Absturz vor Abschluss der Journal-Transaktion

Datenblöcke geschrieben, einige Metadatenblöcke in Journal

Transaktion nicht komplett -> kein Replay, neue Datei nicht vorhanden!



Inode-Block / Verzeichnisblock / Indirect-Block / Bitmap-Block / Datenblock

PERFORMANCE

- **Optimiertes Schreiben:**
  - Lineares Schreiben in Journal
  - Checkpointing mit minimaler Anzahl an Seek-Operationen möglich
- **Minimierung von Write Barriers:**
  - Komplettierung einer Transaktion (kann sehr groß sein: „Compound Transaction“)
  - Nach Checkpointing, mit Komplettierung nachfolgender Transaktionen kombinierbar
- **Leseleistung:** unbeeinflusst von Journaling

IMPLEMENTIERUNG

- Journal orthogonal zu Dateisystem-Layout
- **Speicherort** für Journal:
  - Fester Bereich von Blöcken *(Reiserfs)*
  - Versteckte Datei, vorallokiert *(Ext3, NTFS)*
- **Granularität** der Journal-Einträge:
  - Ganze Metadatenblöcke *(Ext3, Reiserfs)*
    - Nur „sichere“ Versionen von Metadatenblöcken
    - Problemstellung ähnlich wie bei Soft Updates
  - Einzelne Metadaten-Updates *(NTFS)*

JOURNAL-MODE (EXT3)

- **Write-back Journaling:**
  - Nur Metadaten in Journal
  - Datenblöcke „irgendwann“ geschrieben
  - Nicht initialisierte Dateiinhalte möglich
- **Ordered Journaling [Voreinstellung]:**
  - Erst Daten „in-place“ schreiben, dann zugehörige Transaktion in Journal markieren
  - Keine nicht initialisierten Dateiinhalte
- **Data Journaling:**
  - Daten + Metadaten in Journal

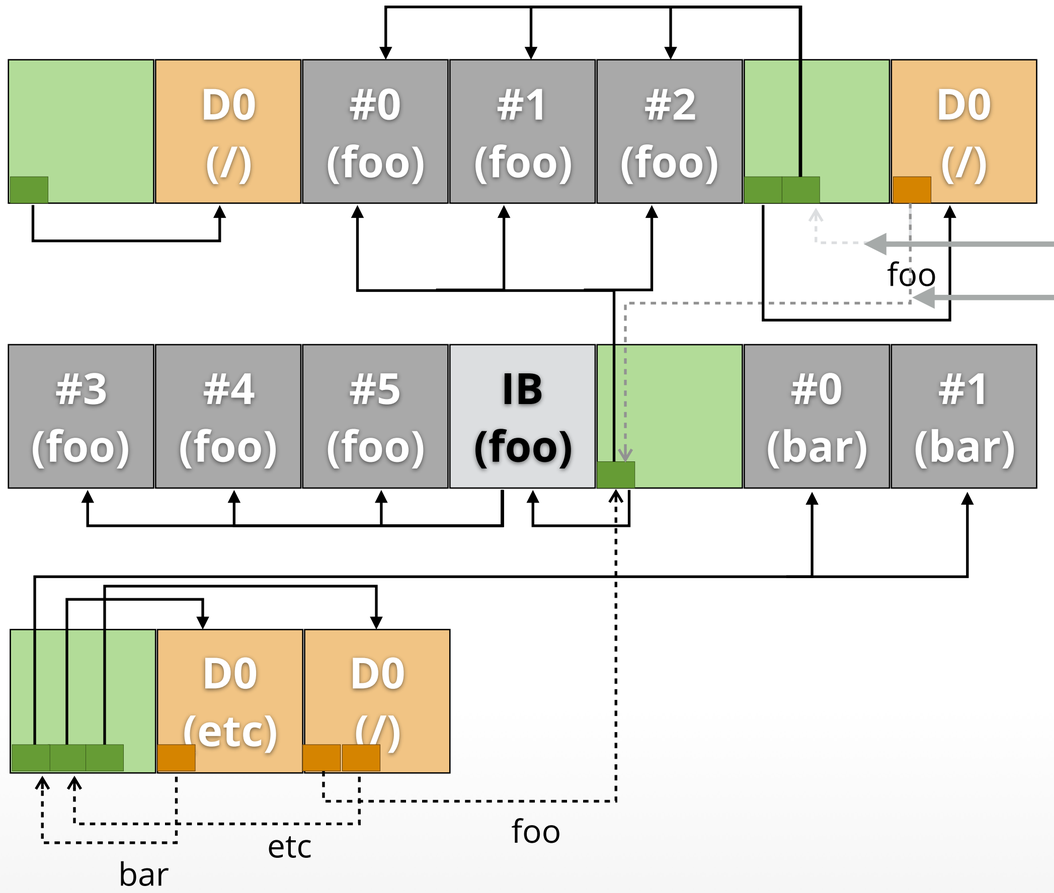
GLIEDERUNG

- Dateisystemstrukturen
- Inkonsistenzen nach Abstürzen
- Konsistenzmechanismen:
  - Synchrones Schreiben
  - Soft Updates
  - Journaling
  - **Log-strukturiert**
  - Copy-on-write / Shadow Paging

DATEISYSTEM ALS LOG

- **Idee:** gesamtes Dateisystem als Log
  - Daten und Metadaten linear geschrieben
  - Kein Überschreiben, neue Blockversionen werden an Log angehängt
  - Jeweils neuste Version aller Blöcke stellen aktuellen Dateisystemzustand dar
- Interessante Eigenschaften:
  - Log enthält alte Zustände des Dateisystems
  - Mehrere **Snapshots** gleichzeitig möglich

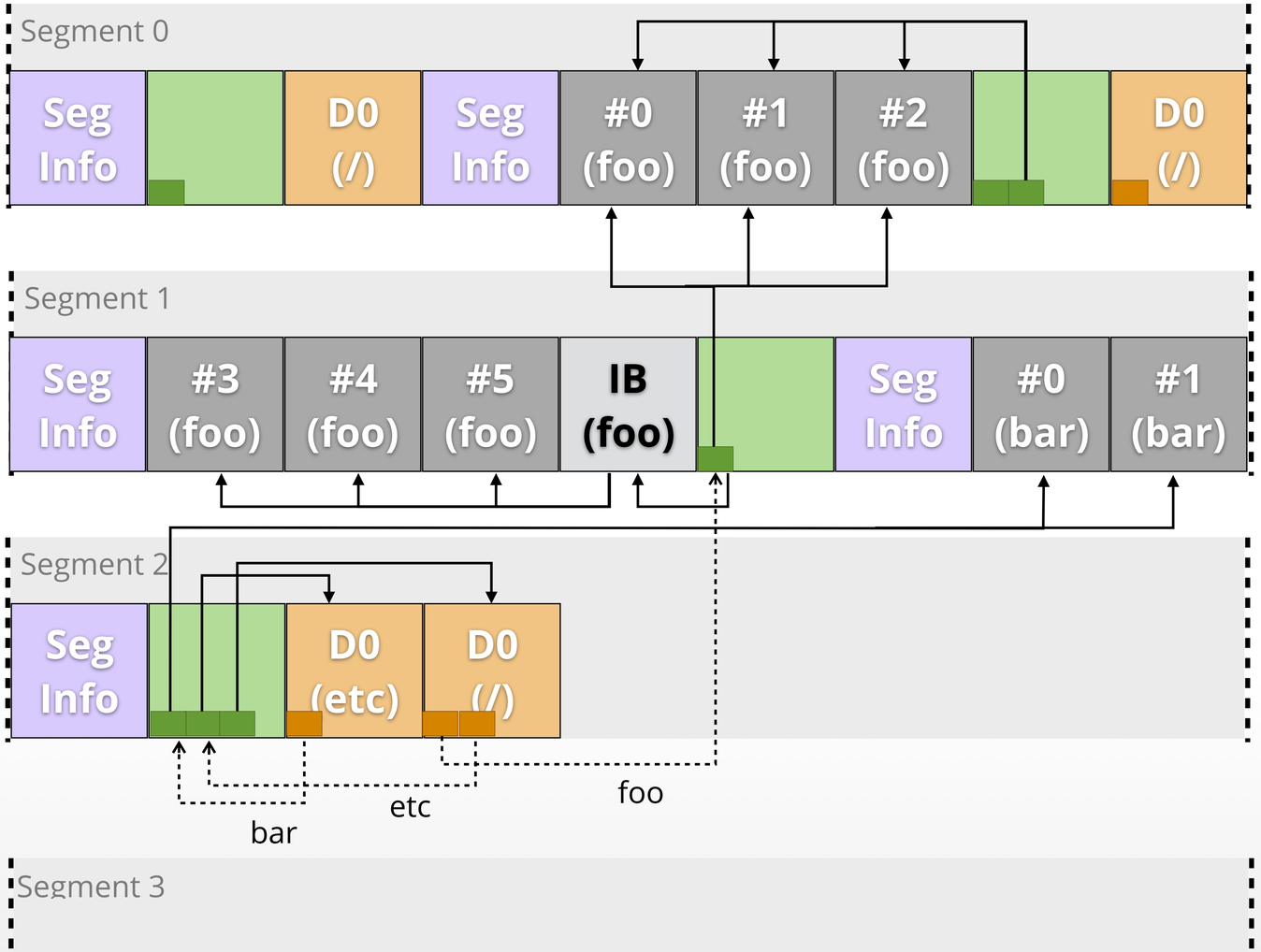
DATEISYSTEM ALS LOG



Frühere Abbildungen des Dateinames foo auf ältere Inode-Versionen

Inode-Block / Verzeichnisblock / Indirect-Block / Datenblock

LOG-SEGMENTIERUNG



Inode-Block / Verzeichnisblock / Indirect-Block / Datenblock / Summary-Block

AUFBAU DES LOGS

- Speichermedium in **Segmente** unterteilt
- Enthalten **Daten-** und **Metadatenblöcke**
- **Summary-Blöcke** beschreiben Blocktypen:
  - Datei-/Verzeichnisblöcke: **(Inode,Block#)**
  - **Inode-Blöcke:** Neue/modifizierte Inodes
  - **Inode-Map-Blöcke:** Inode-Position in Log
- Inode-Map entkoppelt Inode-Zeiger in Verzeichnissen von physischer Position

CHECKPOINTS

- Alle Daten und Metadaten im Log …
  Aber: kompletter Scan bei jedem Start zu teuer
- **Besser:** regelmäßig Zwischenstand der aktuellsten Metadaten sichern
- **2 Checkpoint-Areas** an fester Position:
  - Konsistente Versionen aller Blöcke aus Inode-Map
    (+ Segment-Usage-Tabelle)
  - Zeitstempel + Zeiger auf letztes Segment
  - Markierung / Prüfsumme um Konsistenz bzw. Vollständigkeit zu erkennen

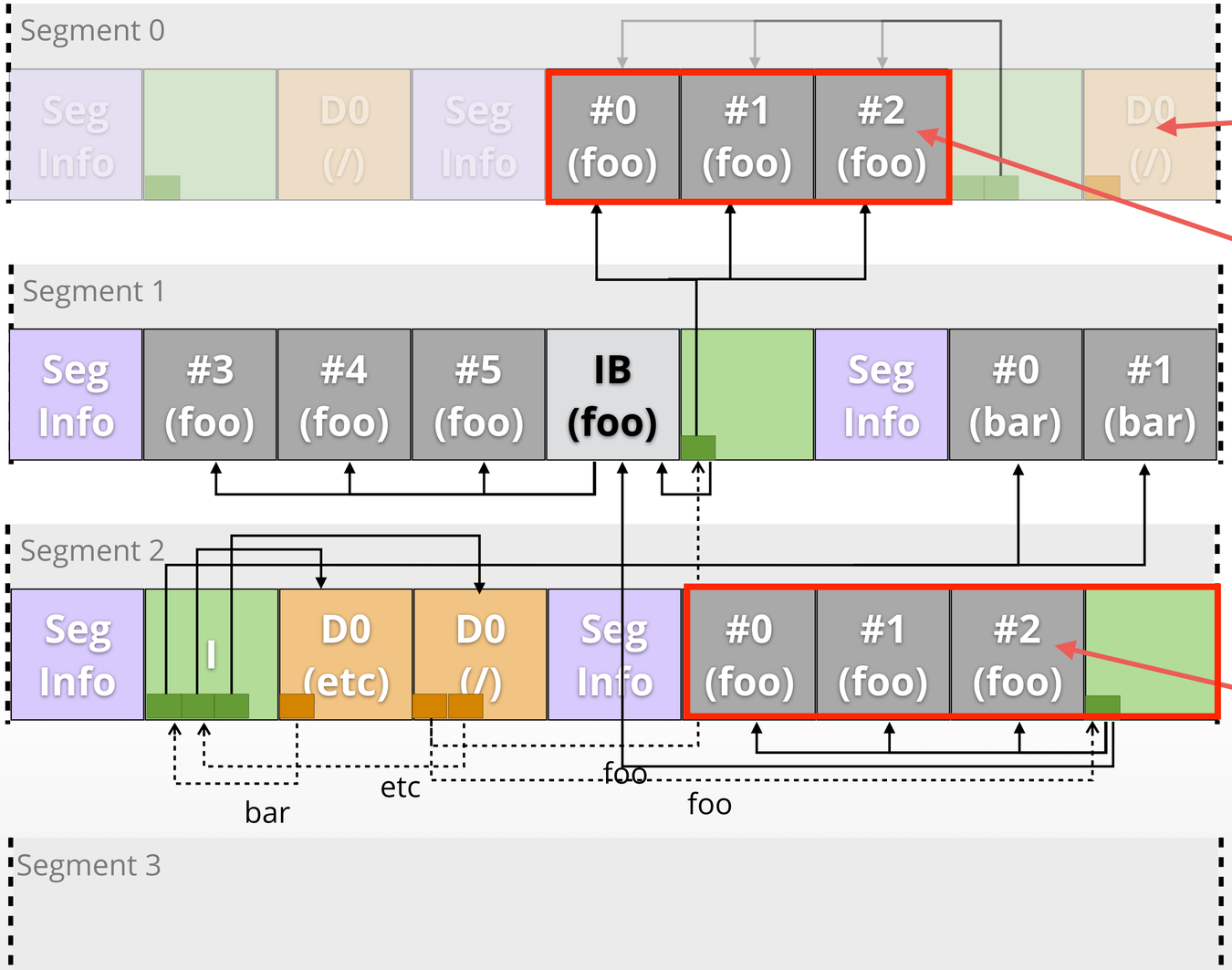
SCHNELLES MOUNTEN

- **Mounten (auch nach Absturz):**
  1. Wähle aktuellste konsistente Checkpoint-Area
  2. Reinitialisiere Inode-Map in Kernelspeicher
  3. Bestimme Position des letzten vor Checkpoint geschriebenen Segments
  4. Roll-forward ab letztem Segment
  5. Aktuelles Dateisystem wiederhergestellt

GARBAGE-COLLECTION

- **Problem:** Log darf nicht unendlich wachsen
- **Lösung 1:** alte Blöcke einzeln freigeben
  - (+) Freigabe ist einfach und schnell
  - (-) Fragmentierung nimmt immer mehr zu
- **Lösung 2:** komplette Segmente freigeben
  - (+) Fragmentierung kaum relevant (Segmente sind groß)
  - (-) Noch gültige Blöcke in Segment müssen in neues Segment gerettet werden
- In Praxis: nur **Lösung 2** (LFS [3] und andere)

SEGMENT FREIGEBEN



Obsolete Versionen von Blöcken

Aktuelle Versionen von Blöcken in einem Segment, das freigegeben werden kann, sobald die Blöcke in ein neues Segment kopiert wurden

In neues Segment gerettete Datenblöcke und neue Inode-Version, die darauf verweist

GARBAGE-COLLECTION

- **Cleaner-Prozess** sucht teilweise belegte Segmente im Hintergrund
- Wird automatisch aktiv, wenn Anzahl freier Segmente Schwellwert unterschreitet
- Identifikation veralteter Blöcke:
  - Suche Inode oder Indirect-Block, der auf untersuchten Block zeigt
  - Kein Zeiger gefunden? -> Freigabe möglich
- Weiterführende Details in [3]

GLIEDERUNG

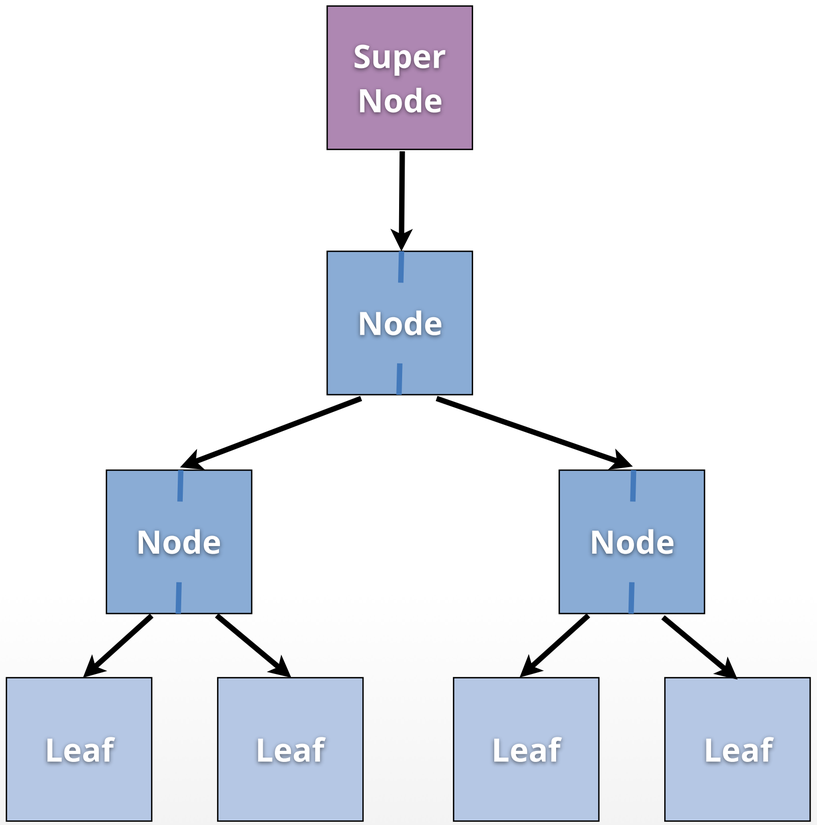
- Dateisystemstrukturen
- Inkonsistenzen nach Abstürzen
- Konsistenzmechanismen:
  - Synchrones Schreiben
  - Soft Updates
  - Journaling
  - Log-strukturiert
  - **Copy-on-write / Shadow Paging**

COPY-ON-WRITE

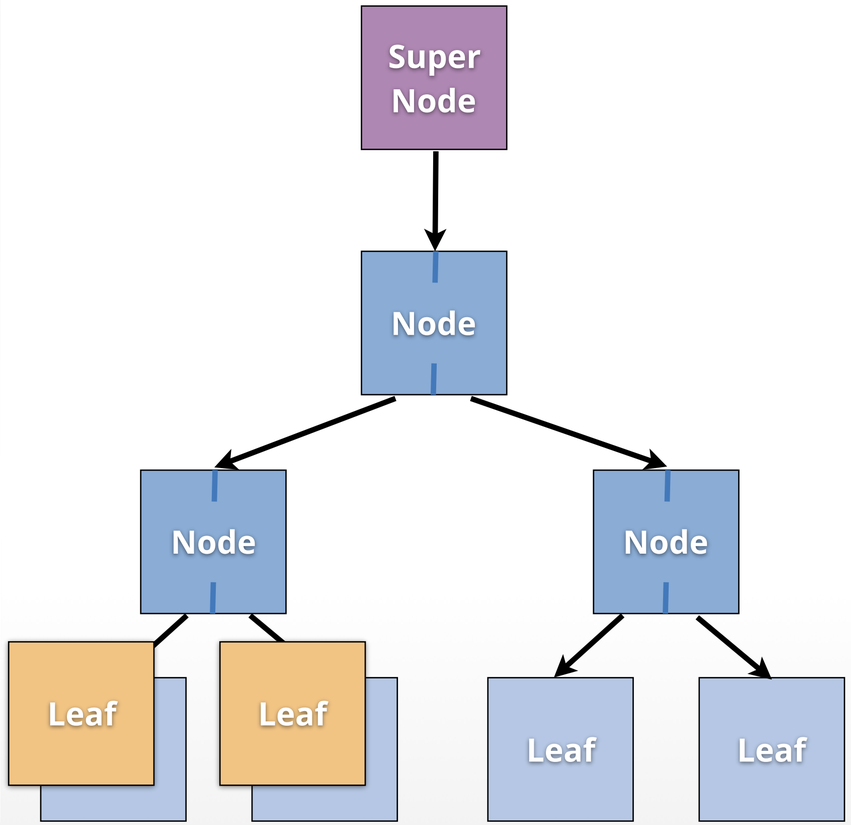
- **Grundidee** ähnlich zu Log-struktierten Dateisystemen: Überschreibe niemals!
- Gesamtes Dateisystem als **B+-Baum** (oder Hierarchie von B+-Bäumen)
- Änderungen an Dateisystemstrukturen:
  - **Copy-on-write:** Modifizierte Version eines **Knoten** (Block) an freie Position schreiben
  - Anpassung der Zeiger in Eltern-Knoten -> Copy-on-write auch für Eltern, bis zur Wurzel
- Aktualisierung des Wurzelzeigers: schaltet **atomar** auf neuen Dateisystemzustand um

COPY-ON-WRITE

(1) Ursprungszustand

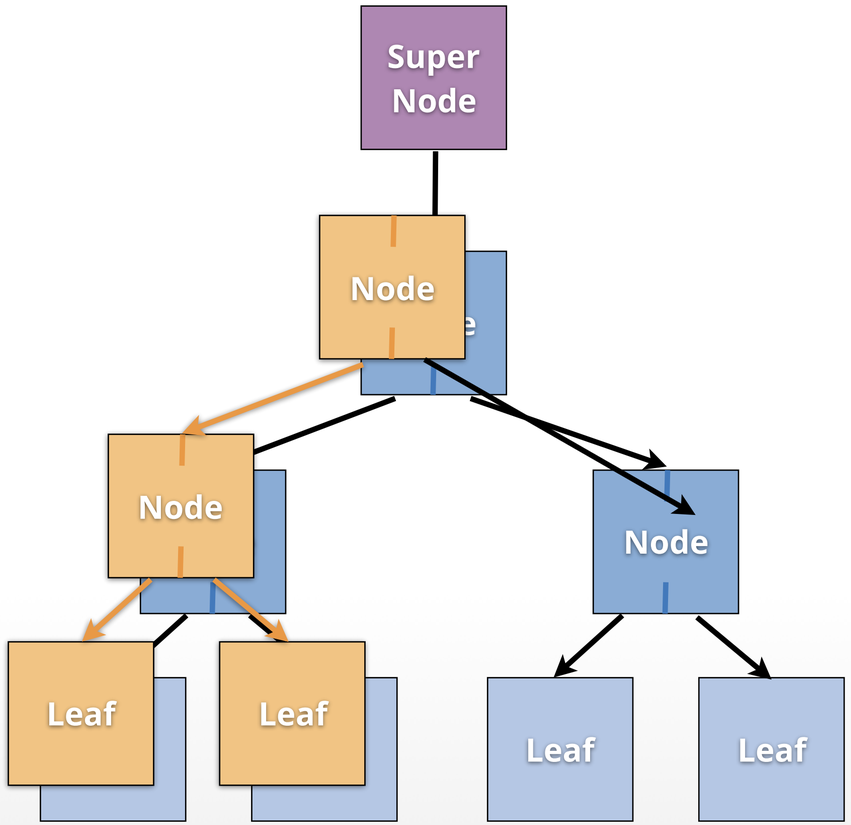


(2) Modifzierte Blöcke geschrieben

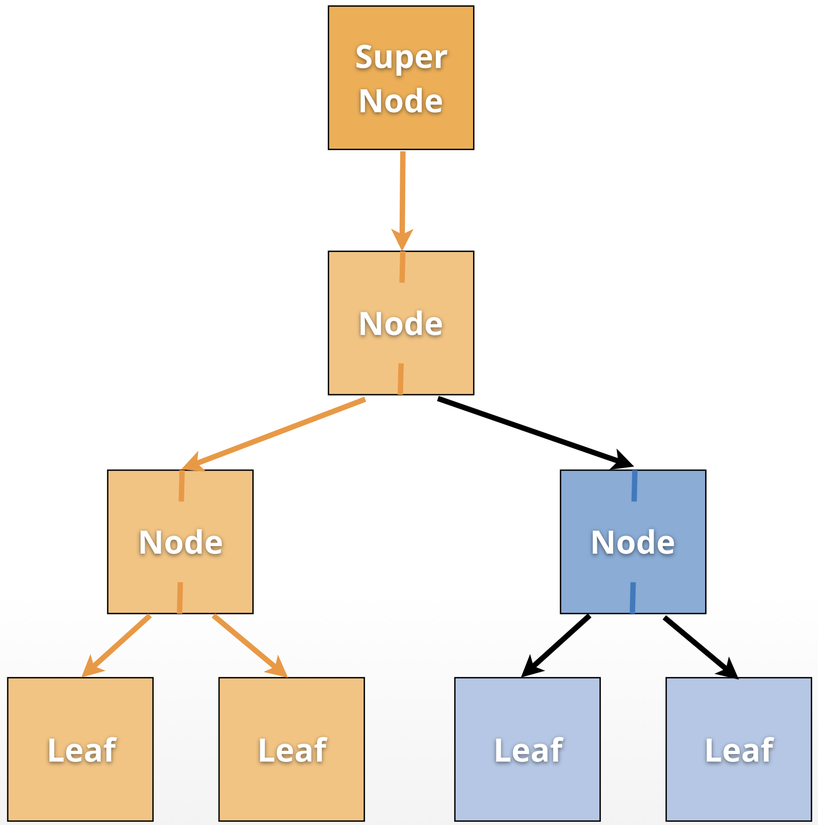


COPY-ON-WRITE

(3) Elternknoten aktualisiert



(4) Wurzelknoten neu geschrieben, alte Blockversionen freigegeben



ÜBERBLICK

- Journaling dominiert: *Ext3/Ext4, XFS, NTFS, …*
- Log-strukturierter Ansatz:
  - Linux: *JFFS2, YAFFS, NILFS2, F2FS*
  - In SSDs: Flash-Translation-Layer (FTL)
- Copy-on-write für große Speichersysteme:
  - Oracle / Linux: *ZFS, BTRFS*
  - macOS: *APFS*
  - Windows 8+: *ReFS* (inzwischen nur noch Server-Version)

VERGLEICH: FS/HDD/SSD

Strategie	Festplatte	Solid State
Synchron	R(+), W(--)	R(++), W(--)
Soft Updates	R(+), W(+)	R(++), W(+)
Journaling	R(+), W(+)	R(++), W(+)
Copy-on-write	R(o), W(+)	R(++), W(++)
Log-strukturiert	R(-), W(++)	R(++), W(++)

Operation: Lesen: **R**, Schreiben: **W**

Leistung: schlecht **-- - o + ++** sehr gut

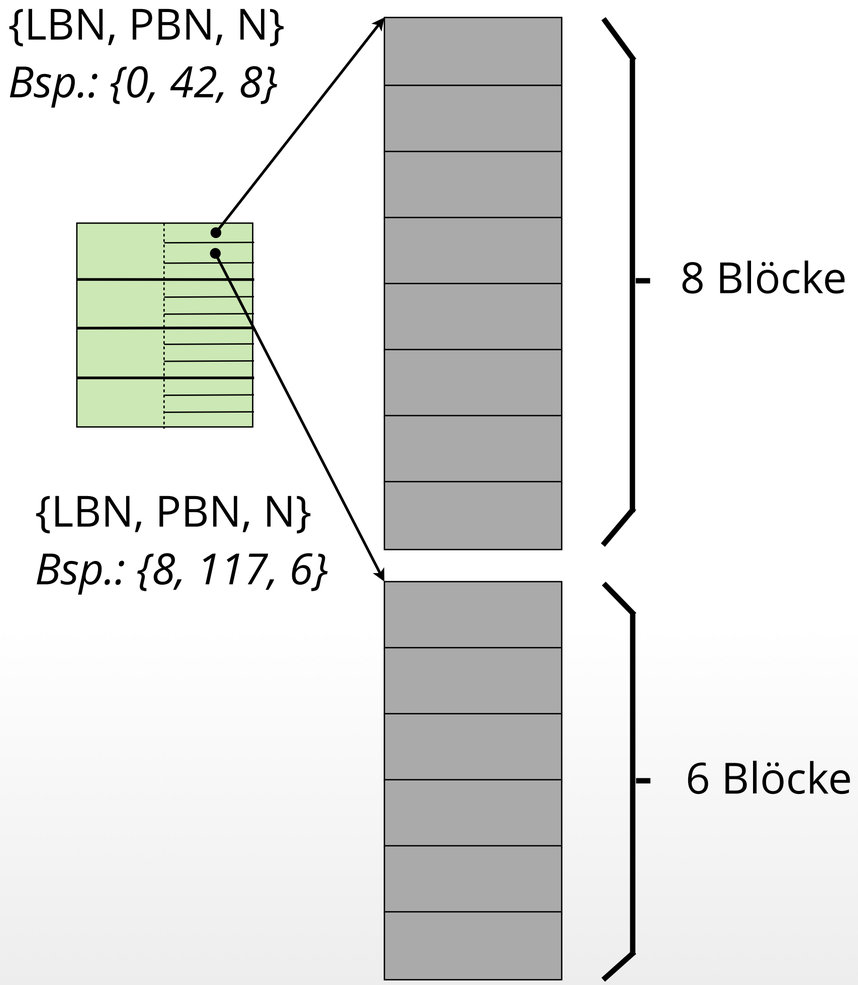
NACHTRAG: DATEI-SYSTEMSTRUKTUREN

OPTIMIERUNGEN

- Sehr kleine Dateien (kleiner als Blockgröße):
  - **Inline-Data:** Speicherung weniger Bytes direkt in Inode (z.B. neu in Ext4)
  - **Tail-Packing:** viele Dateireste in einem Block
- Sehr große Dateien:
  - **Problem:** unnötig viele Blockzeiger
    - Zeiger auf direkt aufeinander folgende Blöcke
    - Hoher Speicherbedarf, Indirect-Blocks
  - **Lösung:** Extents

EXTENTS

- **Extents:** aufeinander folgende Blöcke
  - **LBN:** Logische Position innerhalb der Datei
  - **PBN:** Physische Block-nummer des 1. Blocks
  - **N:** Anzahl Blöcke
- Vorteil: wenige Extent-Deskriptoren anstatt vieler Blockzeiger



EXTENTS IN EXT4

- Beispiel **Ext4:**
  - Maximal 128 MB pro Extent
  - 4 Extent-Deskriptoren pro Inode
  - Sehr große/stark fragmentierte Dateien:
    - Mehrstufige Extent-Bäume (mehr als 4 Blätter)
    - Ähnlich Indirect Blocks, aber Knoten sind Extent-Deskriptoren statt Blockzeiger
- Fast alle modernen Dateisysteme unterstützen inzwischen Extents

QUELLEN

[1] SATA-IO: Native Command Queuing: http://www.sata-io.org/technology/ncq.asp

[2] *„Soft updates: a Technique for Eliminating Most Synchronous Writes in the Fast Filesystem“*, Marshall Kirk McKusick und Gregory R. Ganger, ATEC '99 Proceedings of the annual conference on USENIX Annual Technical Conference, 1999

[3] *„The Design and Implementation of a Log-Structured File System“*, Mendel Rosenblum und John K. Ousterhout, ACM Transactions on Computer Systems (TOCS) Volume 10 Issue 1, Februar 1992