TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

ROBUST FILE SYSTEMS

DISTRIBUTED OPERATING SYSTEMS

CARSTEN WEINHOLD

ROADMAP

- File system structures
- Inconsistencies after a crash
- Consistency mechanisms:
  - Synchronous writes
  - Soft updates
  - Journaling
  - Log-structured
  - Copy-on-write

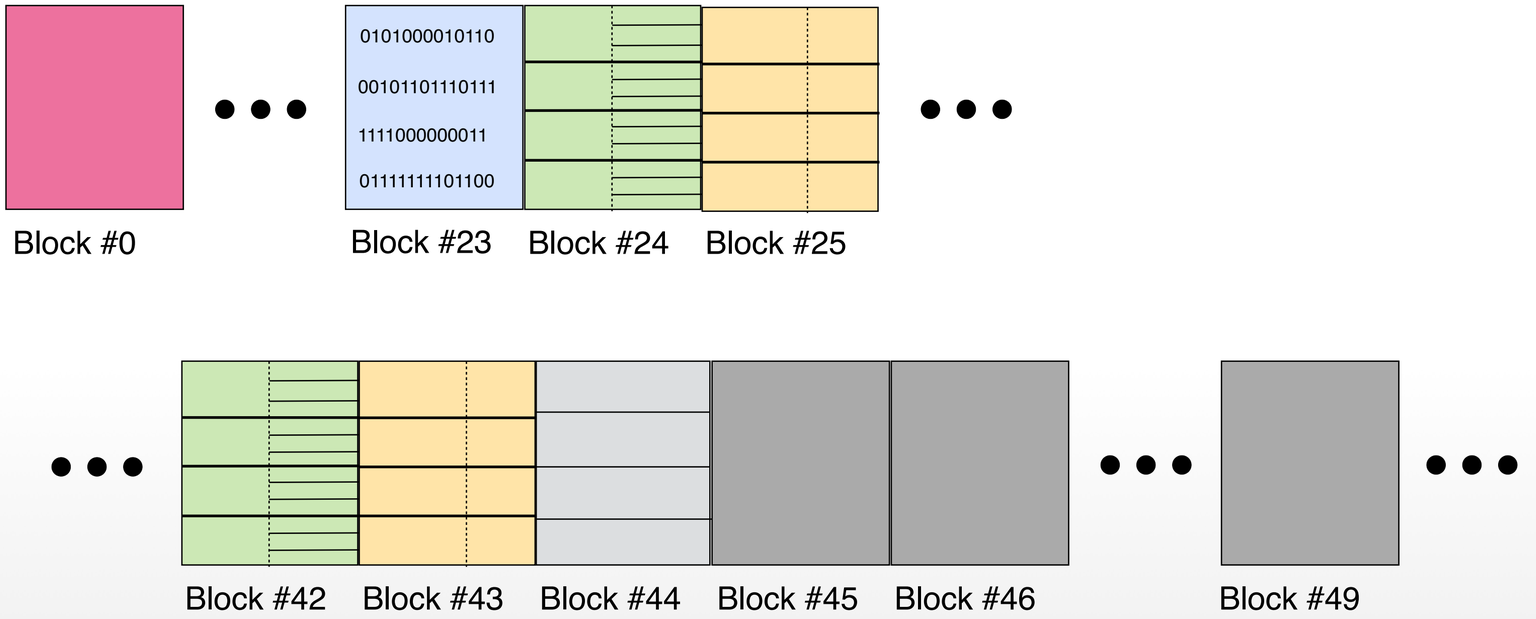
FILE SYSTEMS

- File system maps objects (files) to locations (blocks)
- Mapping described by metadata:
  - Hierarchy of directories: dir entries map filenames to inodes
  - Inodes store attributes and pointers
  - Pointers specify blocks that store file contents
  - Status of inodes, blocks: free / used
- All metadata is stored in (metadata) blocks

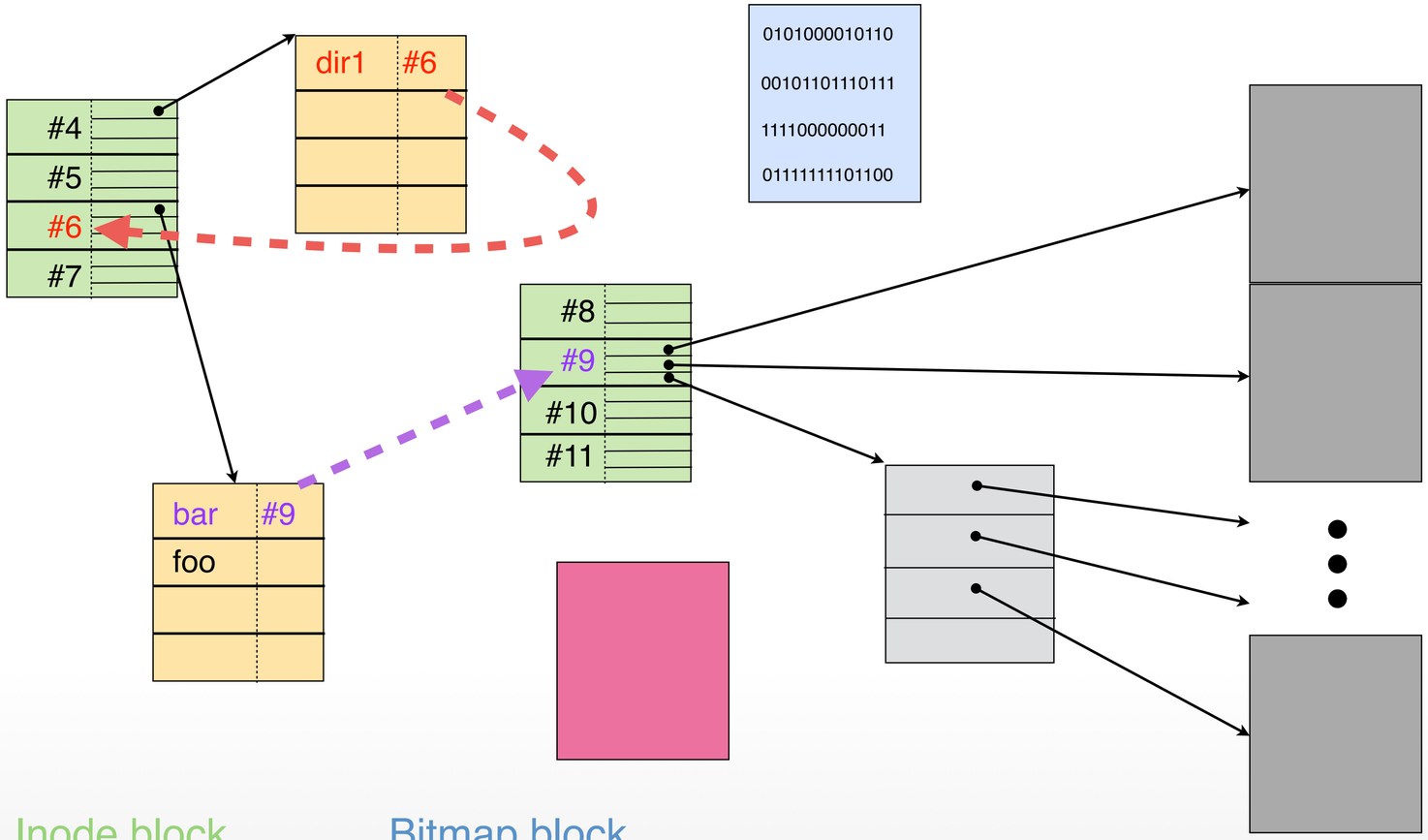
FILE SYSTEM STRUCTURES

Blocks on storage medium are addressed linearly by block numbers

File system structures are read and written as whole blocks



FILE SYSTEM STRUCTURES



Inode block
Directory block
Indirect block
Bitmap block
Data block
Super block

File system structures for filewith pathname “…/dir1/bar”

ROADMAP

- File system structures
- Inconsistencies after a crash
- Consistency mechanisms:
  - Synchronous writes
  - Soft updates
  - Journaling
  - Log-structured
  - Copy-on-write

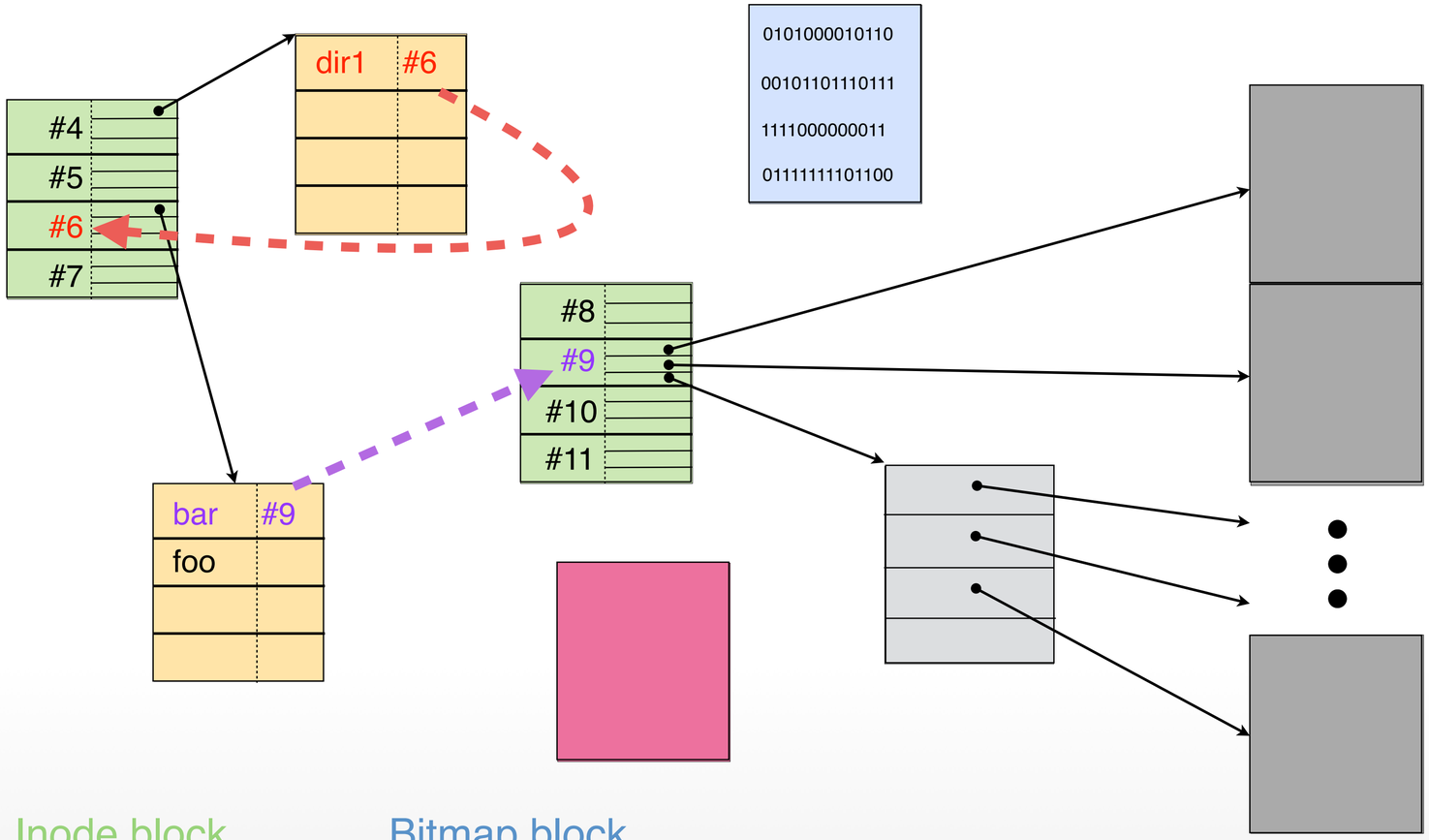
CONSISTENCY PROBLEM

- All changes happen in buffer cache
- Dependencies between changed and unchanged blocks:
  - Metadata and file contents
  - Within metadata

➡ Dependencies also exist between blocks:

- Problem: writing *multiple* blocks is not atomic (usually only sectors can be written atomically)
- Crashes or power failure while writing multiple interdependent blocks can cause **inconsistencies**!

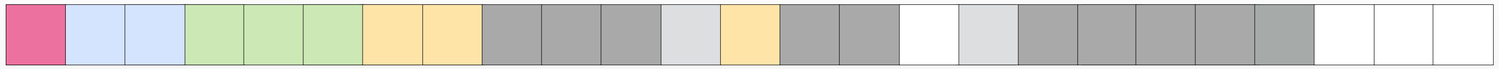
FILE SYSTEM STRUCTURES



Inode block
Directory block
Indirect block
Bitmap block
Data block
Super block

BUFFER CACHE

Any type of access (read/write) to block contents can only happen, if the block contents are in main memory. Therefore, blocks must usually first be read from the storage medium. The operating system manages all block copies in memory in the **buffer cache**.



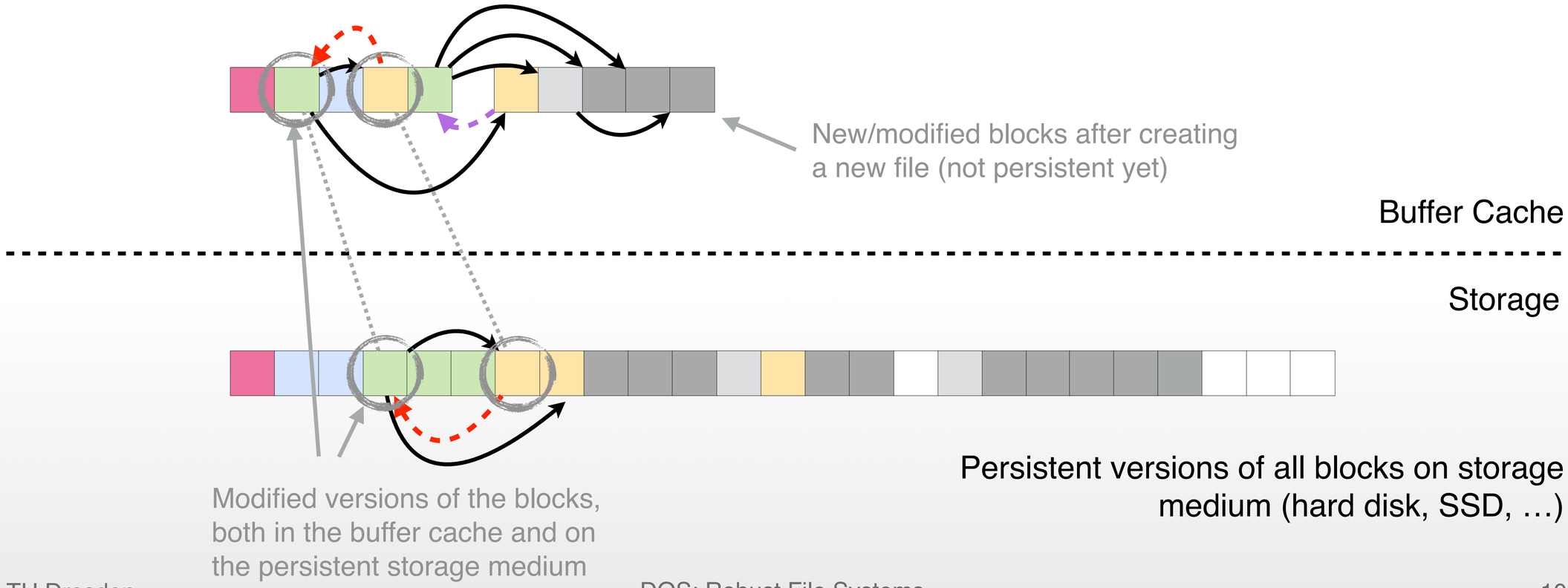
Storage

Persistent versions of all blocks on storage medium (hard disk, SSD, …)

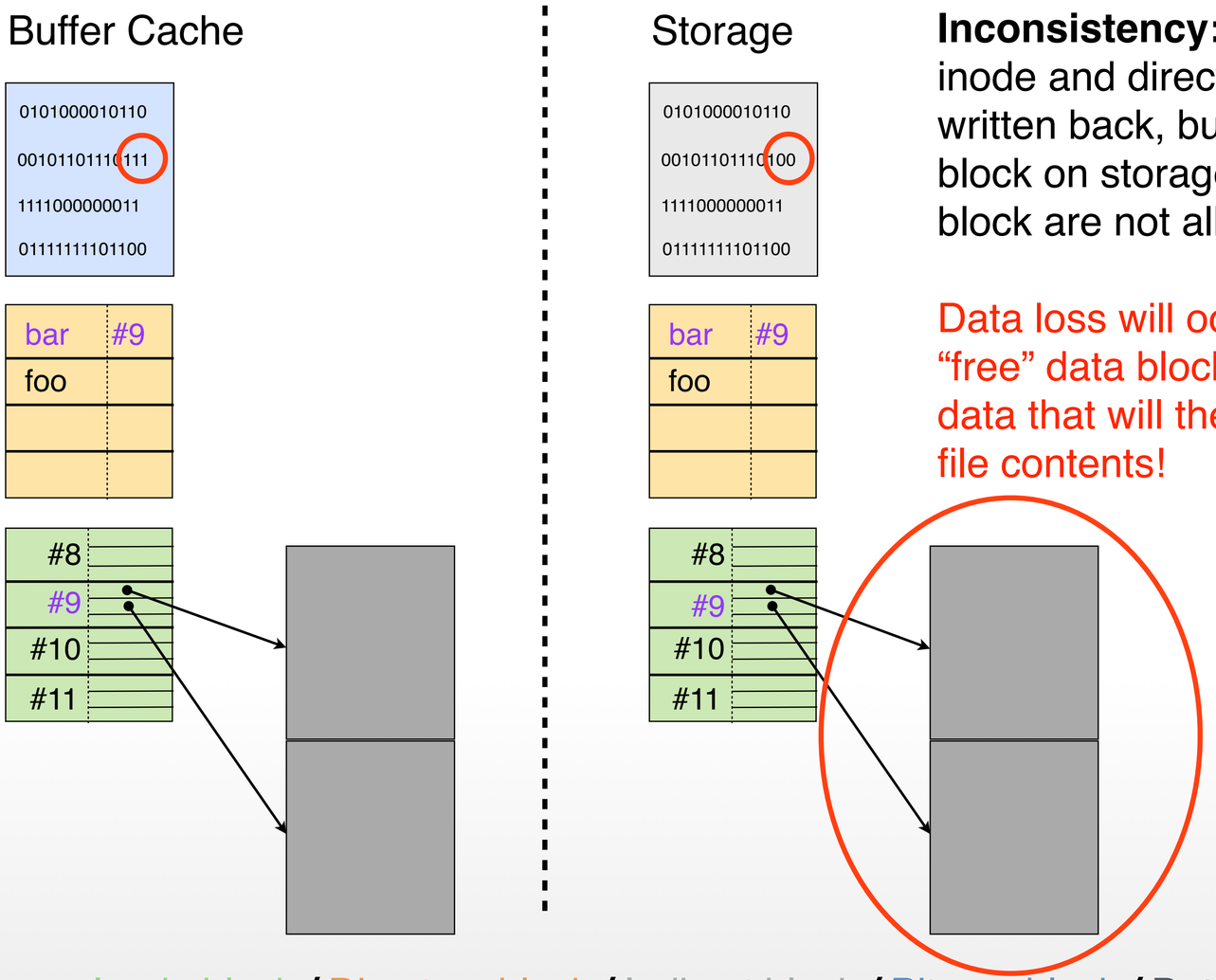
BUFFER CACHE

All changes to blocks are first made to the copies in the buffer cache. These changes must then be written back to the storage medium (**sync** operation).

If only a subset of the modified blocks is written back from the buffer cache, **inconsistencies** may occur in the **persistent block copies** on the storage medium!



INVALID ALLOCATION BITMAP



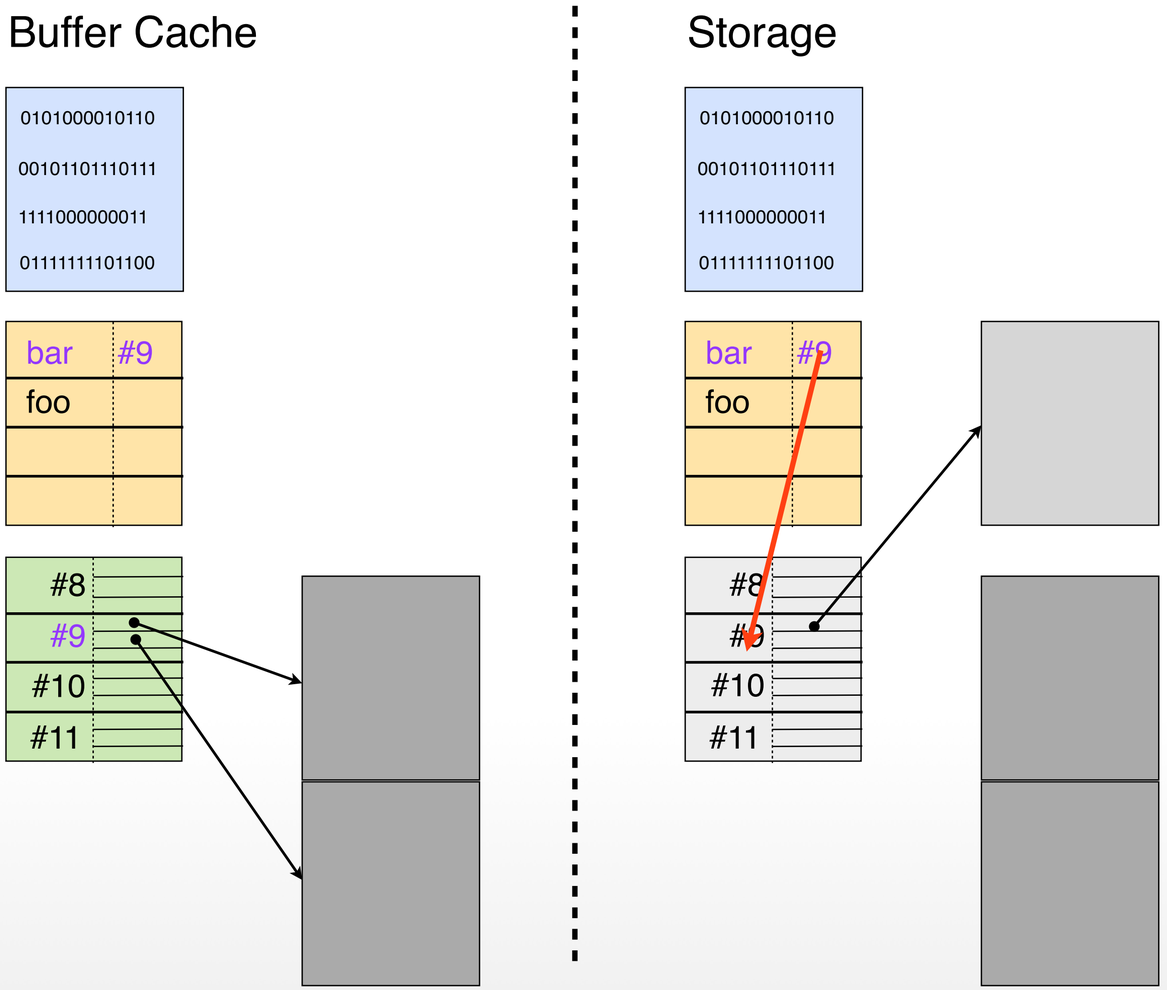
Inconsistency: New versions of data, inode and directory blocks have been written back, but old version of bitmap block on storage medium claims data block are not allocated.

Data loss will occur, when supposedly “free” data blocks are allocated for new data that will then overwrite the existing file contents!

Crash!
(before writing bitmap block)

Inode block / Directory block / Indirect block / Bitmap block / Data block

INVALID POINTERS



Inkonsistency: Bitmap, directory and data blocks have already been written, but inode block has not been updated yet.

Crash!
(before writing inode block)

Data from other files or even deleted file contents are referenced by the old version of the inode!

“Free”/“used” state in block allocation bitmap likely does not match the actual blocks referenced from the inode.

Inode block / Directory block / Indirect block / Bitmap block / Data block

TYPES OF INCONSISTENCIES

- **Critical:** loss of already persistent data
  - Pointer to incorrect inode; blocks from deleted / unrelated files
  - Block / inode marked as free, but referenced
  - Value of reference counter in inode too low
- **Non-critical:** temporary resource leaks
  - Free block / inode marked as occupied
  - Reference counter in inode too high
  - Data block *(or inode)* written, but block pointer *(or dir entry)* not
  - Can be repaired 😊

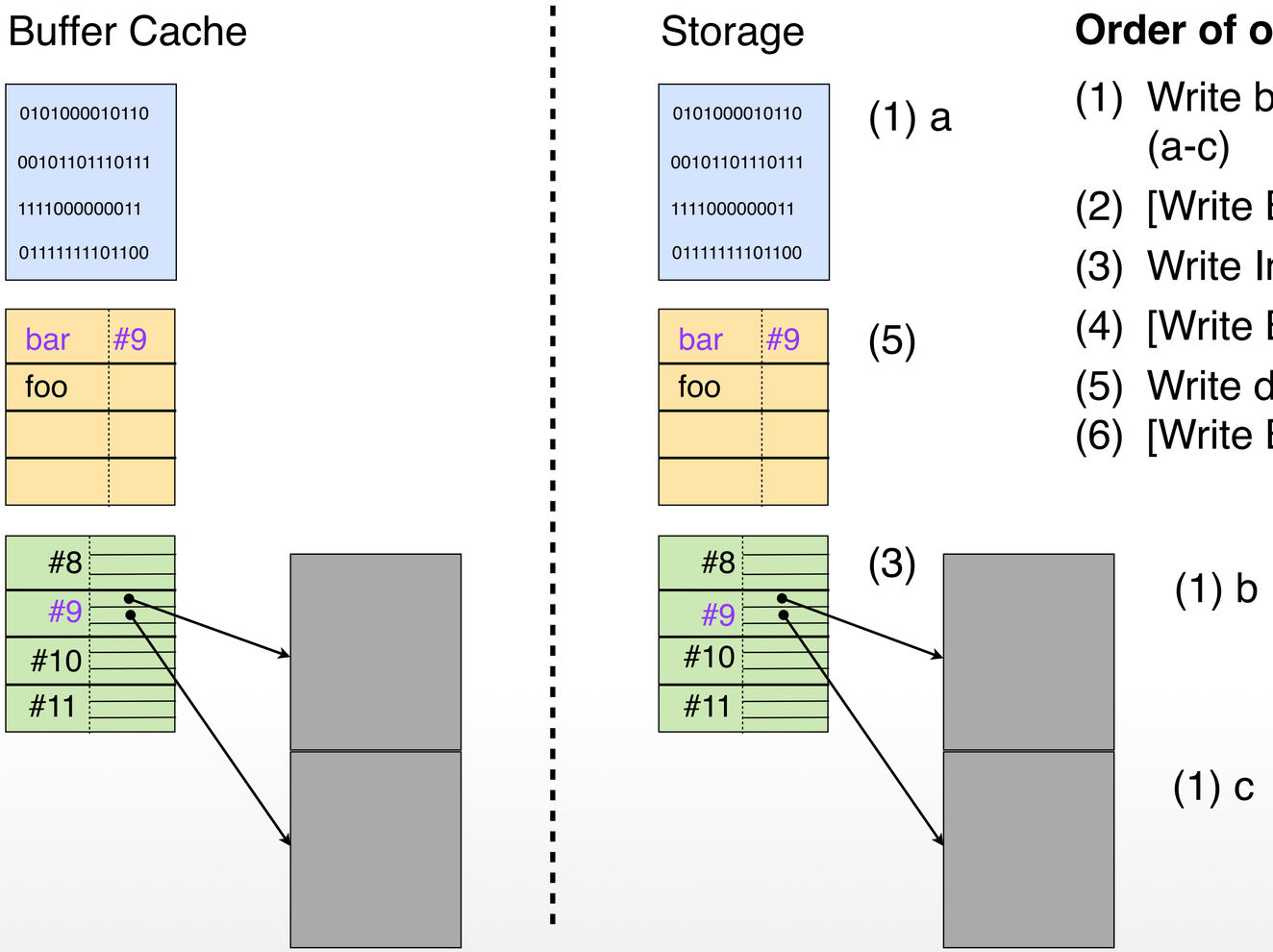
ROADMAP

- File system structures
- Inconsistencies after a crash
- Consistency mechanisms:
  - Synchronous writes
  - Soft updates
  - Journaling
  - Log-structured
  - Copy-on-write

SYNCHRONOUS WRITES

- **Idea:** Write modified blocks immediately in a safe sequence:
  - First write allocation bitmap block, then write new block
  - First write block, then write pointer to it
- Blocks are written **synchronously:**
- Metadata blocks: Wait for feedback from storage device after writing each metadata block (write barrier)
- Data blocks: Waiting for last block to be written is sufficient

SYNCHRONOUS WRITES



Order of operations:

(1) Write bitmap and data blocks (a-c)
(2) [Write Barrier]
(3) Write Inode block
(4) [Write Barrier]
(5) Write directory block
(6) [Write Barrier]

Inode block / Directory block / Indirect block / Bitmap block / Data block

SYNCHRONOUS WRITES

Example: creating a new, empty file

1) Allocate inode, write inode bitmap

[Write Barrier]

2) Initialize inode, write inode block

[Write Barrier]

3) Write inode block:

a) If necessary: allocate new directory block, write bitmap blocks, do necessary write barriers, etc. (not discussed here)

b) Write new/modified directory block

[Write Barrier]

c) Update inode of directory (timestamps, size) +
Write updated inode block

[Write Barrier]

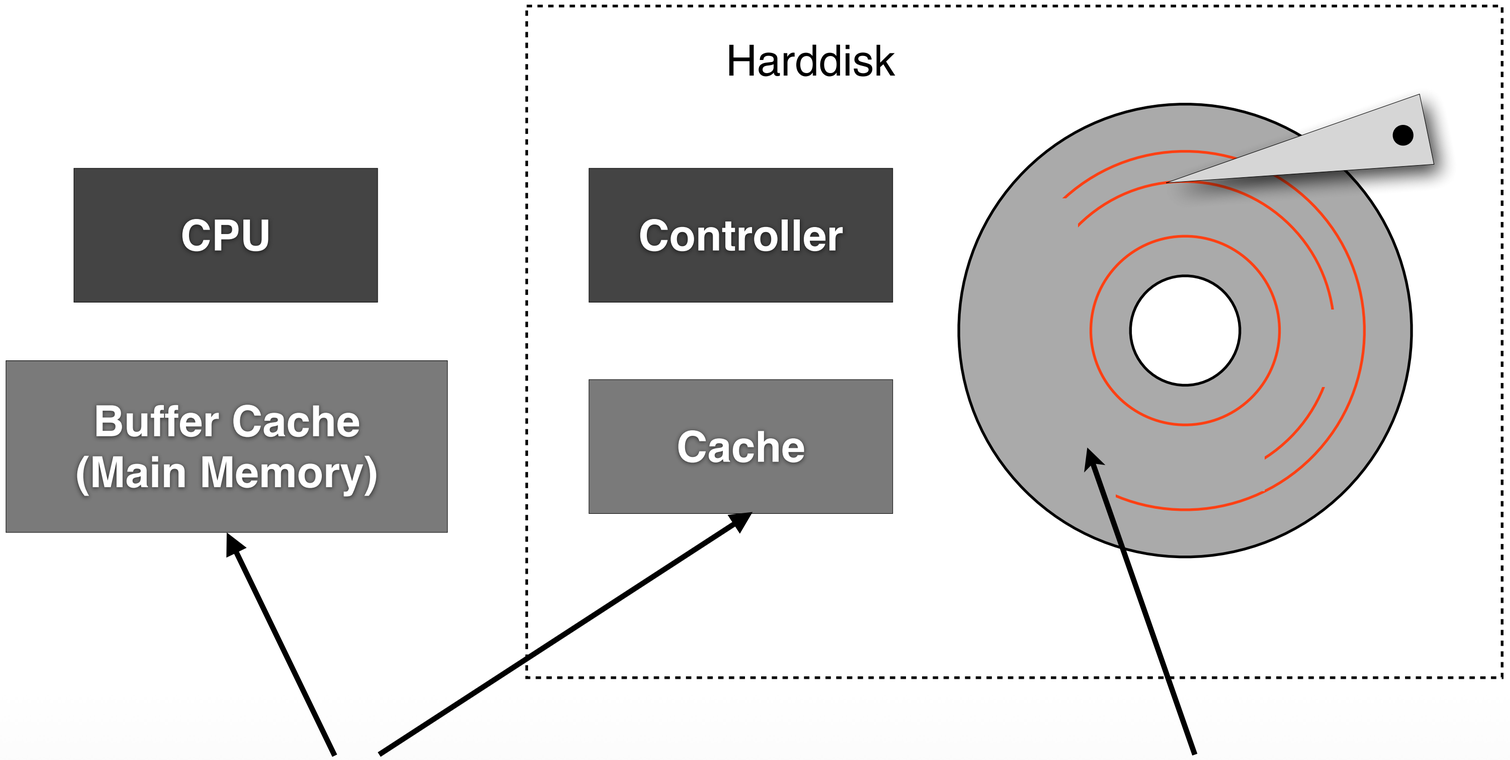
CONSISTENCY RULES

- Fundamental consistency rules:
  - **Valid pointers:** Never write a pointer to a resource, before this resource has been fully initialized
  - **Reuse:** Never reuse the storage location of a deleted resource, before all previous pointers to it have been invalidated
  - **Reachability:** Never invalidate the pointer to a resource that should not be deleted, before the new pointer to this resource has been written

PROPERTIES

- **Only non-critical inconsistencies after crash:**
  - Resource leaks possible: inodes, blocks
  - Values in reference counters too high
  - But: no pointers to uninitialized metadata or invalid file contents
  - **CLEAN** flag not set in superblock
- **Correction:** File system check **(fsck)**
- **Poor performance:**
  - For each operation: several write barriers (expensive!!!)
  - After crash: (very) time-consuming fsck run

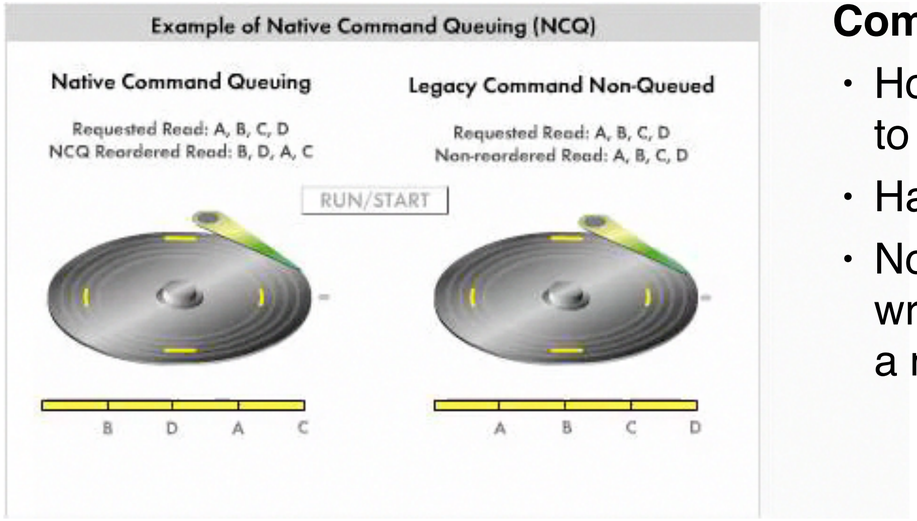
INCONVENIENT HARDWARE TRUTHS



Caches: Operating system manages buffer cache; storage device has its own cache (typically stores whole tracks in case of a harddisk).

Storage media: Data and metadata blocks are distributed across surface (for a harddisk, the read/write head needs to be positioned above the right track).

INCONVENIENT HARDWARE TRUTHS



Source: [1]

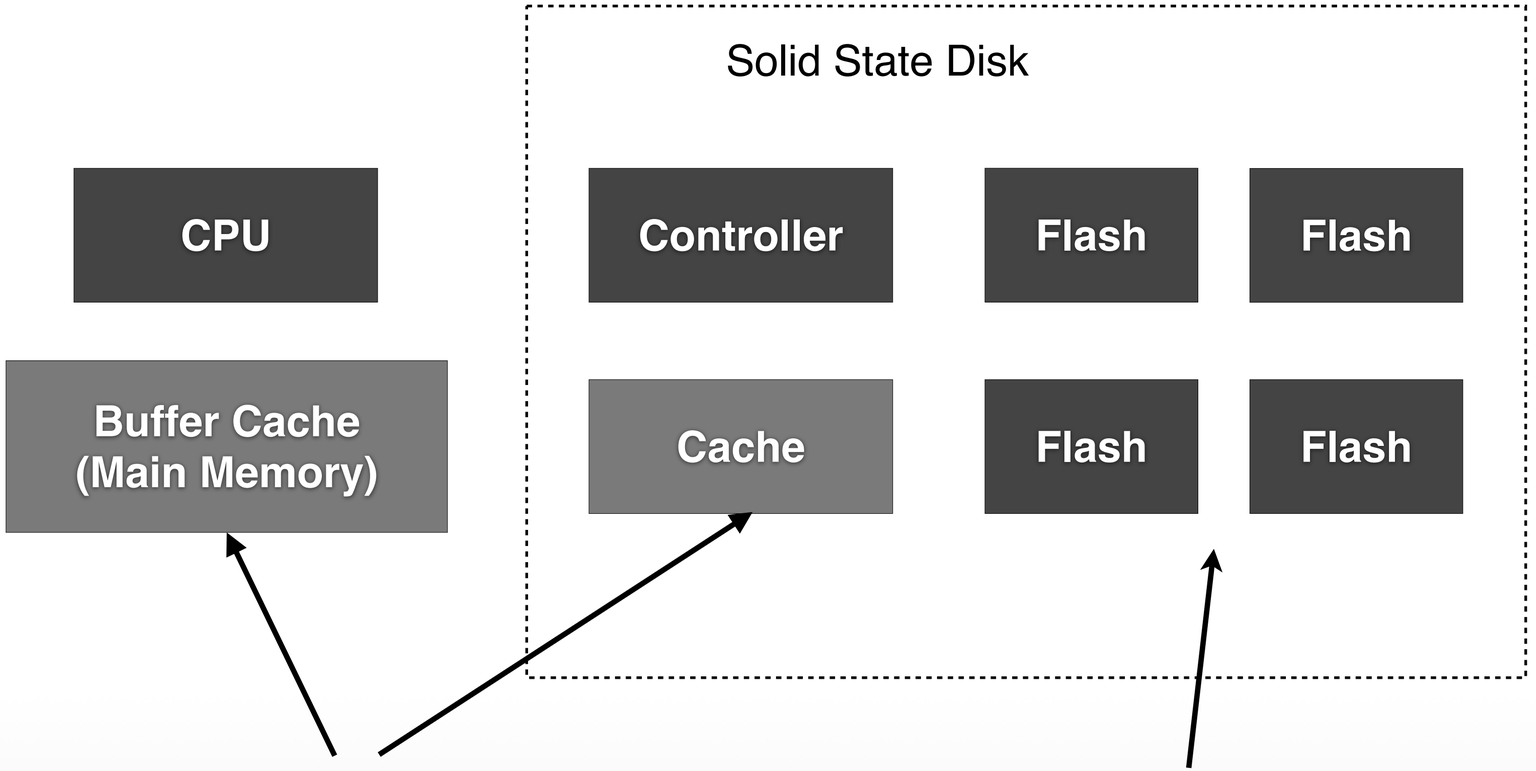
Command queuing:

- Host computer can send multiple read/write jobs to the harddisk
- Harddisk controller decides on optimal order
- No assumptions can be made about the order of writes to persistent storage (but host will receive a notification once requests have completed)

Ensuring a specific order of writes can be difficult:

- Older specifications such as Serial ATA do not recognize write barriers
  → storage device may reorder pending write operations at will
- With Serial ATA, guaranteed persistence of one block before another can only be ensured by explicitly flushing the internal cache (FLUSH command)
- Advantages of command queuing are lost
- New interfaces such as NVMe allow the host to specify write barriers between queued write commands that the storage device must not violate

INCONVENIENT HARDWARE TRUTHS



Caches: Operating system manages buffer cache; storage device has its own cache (typically stores whole tracks in case of a harddisk).

Storage media: Data and metadata blocks are distributed across one or more flash memory chips. Command queuing needed to utilize all flash chips in parallel for maximum performance.

ROADMAP

- File system structures
- Inconsistencies after a crash
- Consistency mechanisms:
  - Synchronous writes
  - Soft updates
  - Journaling
  - Log-structured
  - Copy-on-write

SOFT UPDATES

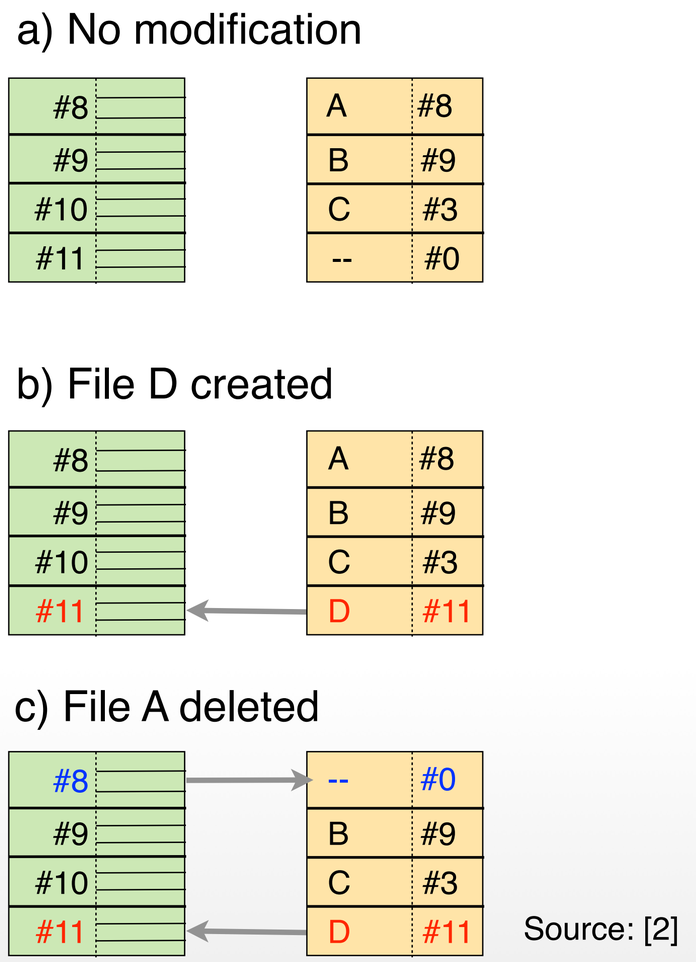
- **Idea:** Buffer many modifications, write in batches (a bit) later
  1. Identify “safe-to-write” blocks (i.e., not causing inconsistencies)
     Note: these blocks are the ones that would be written before the first write barrier when using the “synchronous writes” approach
  2. Write “safe-to-write” blocks in one batch
  3. Do one write barrier
  4. Additional blocks will now be “safe-to-write”, because their dependencies have been satisfied by the previous batch write
  5. Go back to step 1

SOFT UPDATES

- **Implementation details:**
  - Dependency structures for each block in buffer cache:
    - Metadata (e.g., directory → inode)
    - Data (inode / indirect block → data block)
  - Consider all dependencies for all modified blocks
  - Cycles in dependency graph prevent certain blocks from reaching “safe-to-write” state
  - Roll-back conflicting changes to break cycle; write block twice:
    1) “Safe-to-write” intermediate version with only safe changes
    2) Version will previously unsafe (now safe) changes re-applied

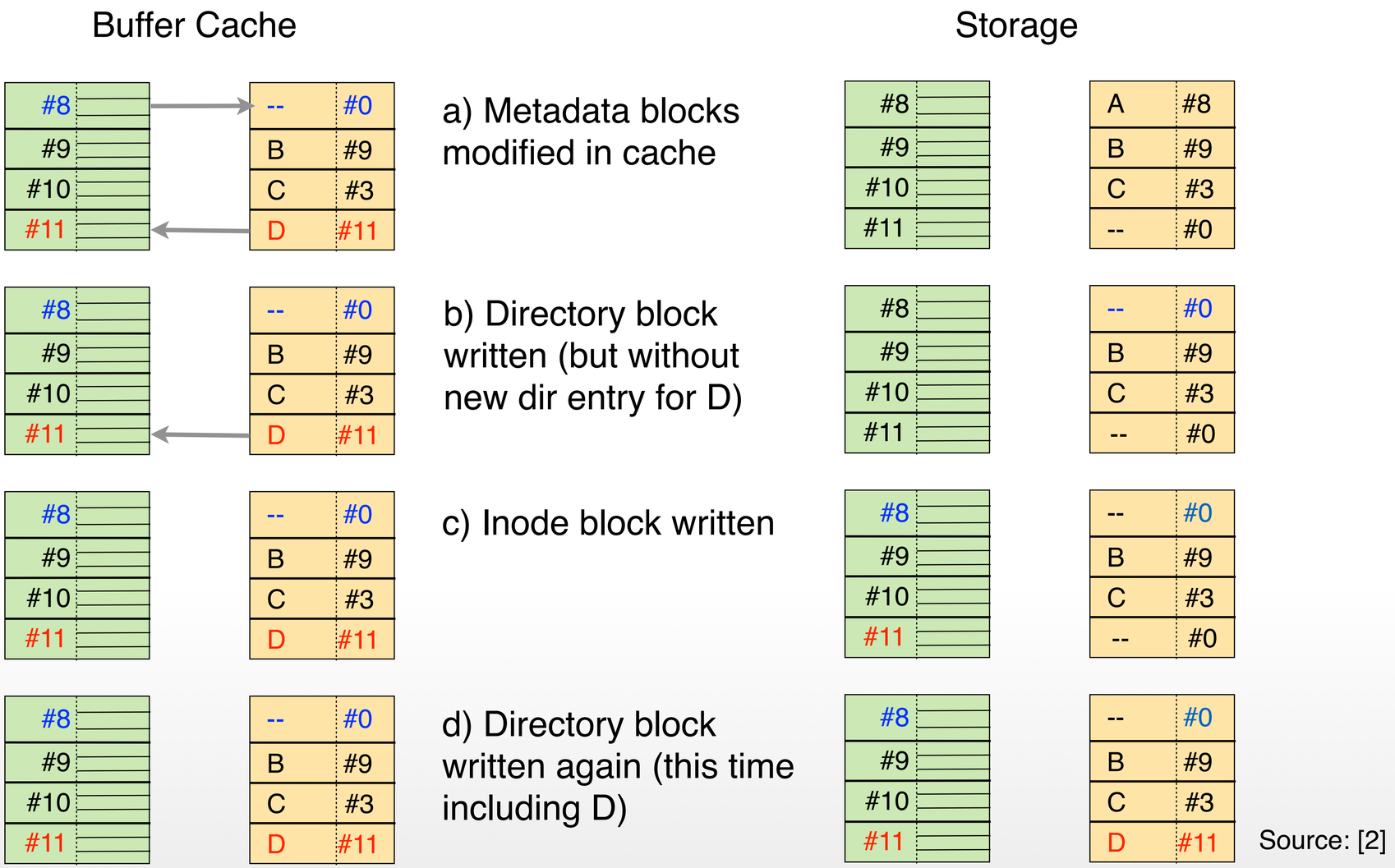
CYCLIC DEPENDENCIES

- State of inode and directory blocks in buffer cache:
  a) No write dependency
  b) Inode block must be initialized before writing directory block
  c) Inode pointer in directory entry must be invalidated before inode can be freed
- Now there is a **cyclic dependency**!



Source: [2]

BREAKING THE CYCLE



Source: [2]

TRACKING DEPENDENCIES

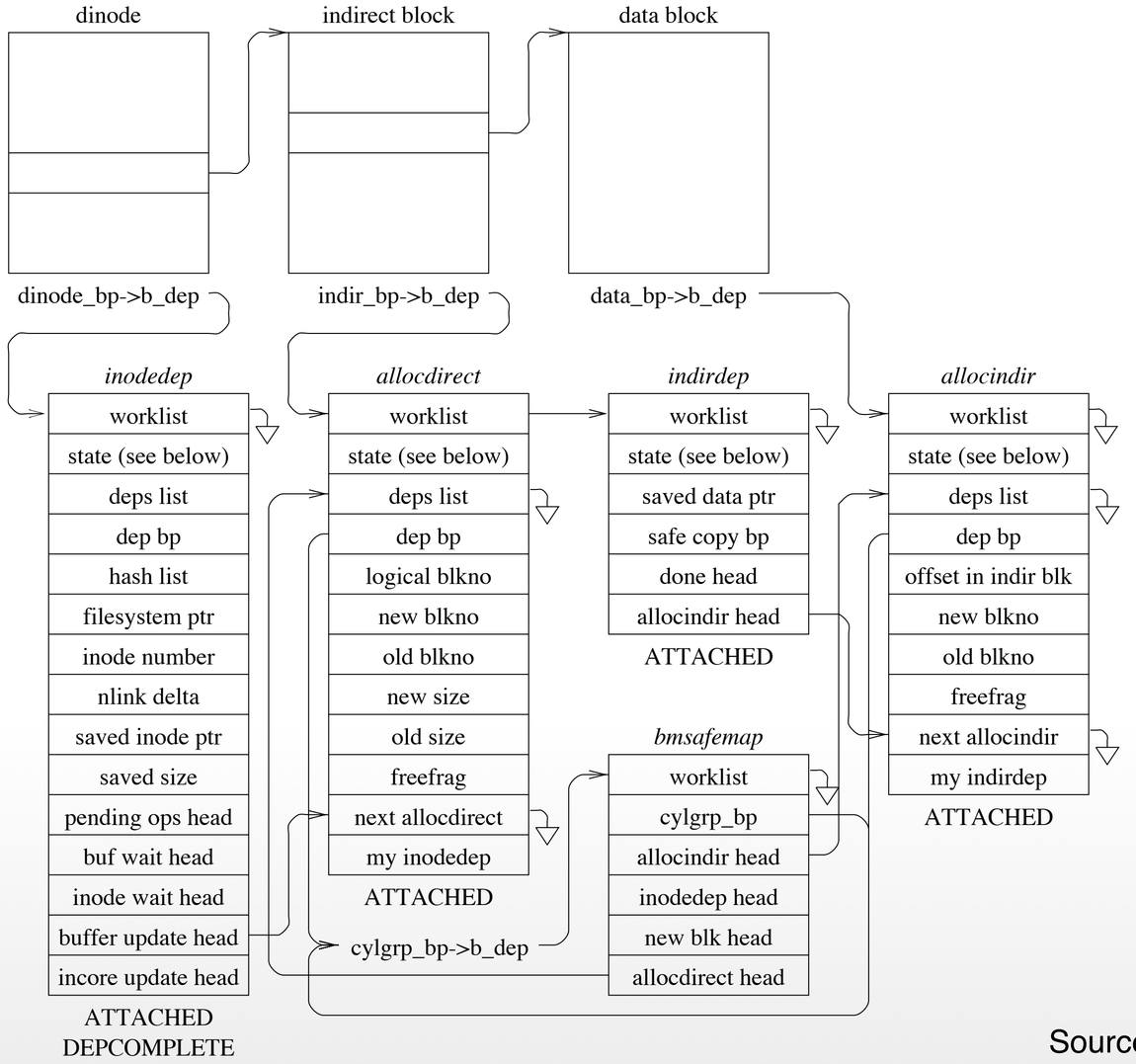
- **Goal:** Write modified blocks at any time
- Soft-updates code inspects dependency structures before write operation:
  - Consistent / isolated modifications can be written immediately without side effects
  - Modifications with unsatisfied dependencies are rolled back temporarily for block-write operation
  - After consistent version of block is been written, re-apply temporarily removed modifications
- There is a dependency structure for each modification in a block (stored in linked list attached to buffer head)

DEPENDENCY STRUCTURES

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

Source: [2]

DEPENDENCY STRUCTURES

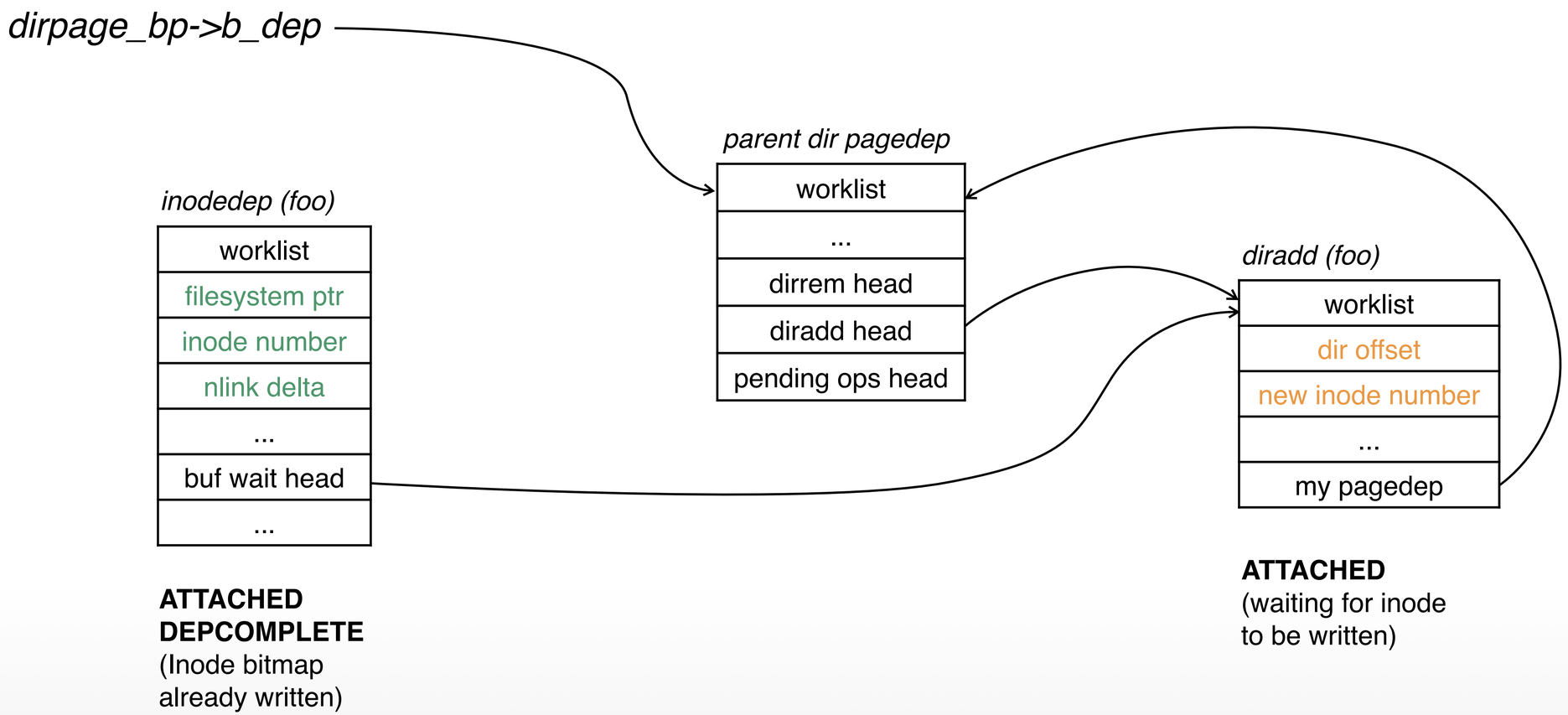


Source: [2]

DEPENDENCY STATE FLAGS

- **ATTACHED:**
  - Buffer is not being written at the moment
  - For rollback: clear ATTACHED flag to indicate that modification must be re-applied after write
- **DEPCOMPLETE:**
  - Modification safe for writing, no need to roll back
- **COMPLETE:**
  - Modification has been written to disk
- Deallocation: only after all flags are set

DEPENDENCY STRUCTURES



Example: dependencies for creating a new directory entry (simplified version of presentation in [2])

PROPERTIES

- **After crash:** only non-critical inconsistencies
  - Free inodes / blocks marked as allocated
  - Reference count in inodes may be too high
  - Both old and new name (hardlink), if crash interrupted a rename operation
- **Correction:**
  - **fsck** to reclaim resources that are incorrectly marked as as used
  - **fsck** can be done in background (if implemented that way)

PROPERTIES

- **Good performance:**
  - Number of write barriers significantly reduced
- **High complexity:**
  - Deeply integrated into file system code and buffer cache
  - Implementation specific to on-disk file system structure
  - Dependency tracking is complicated
  - Roll-back and roll-forward of changes is complicated

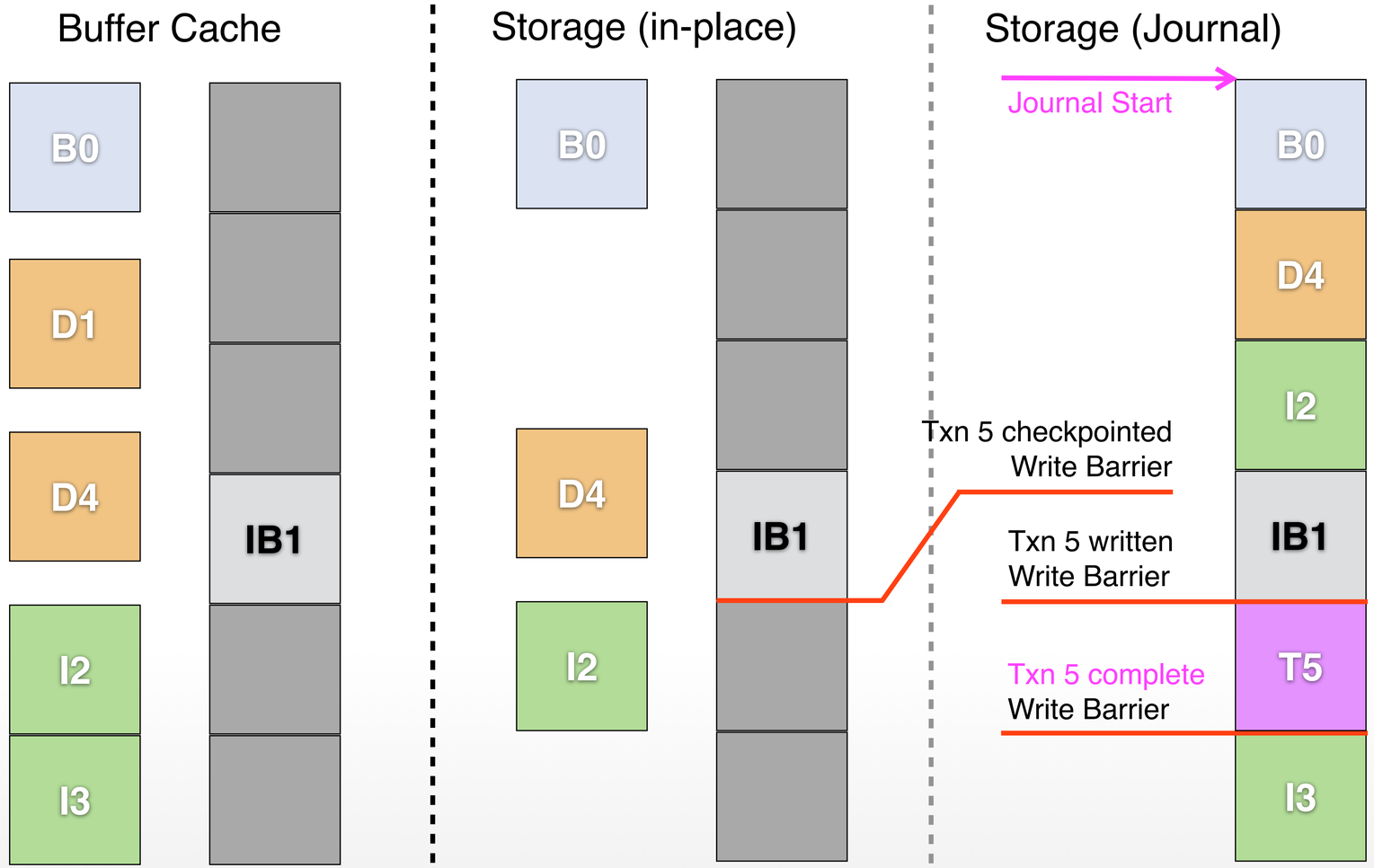
ROADMAP

- File system structures
- Inconsistencies after a crash
- Consistency mechanisms:
  - Synchronous writes
  - Soft updates
  - Journaling
  - Log-structured
  - Copy-on-write

JOURNALING

- **Idea:** write-ahead log (journal)
  1. All changes to file system structures logged in **journal** before actually doing them
  2. Mark journal **transaction** as complete
  3. Write modified blocks to their “in-place” location in the file system (called **checkpointing**)
  4. Release transaction in journal
- **After crash:** read journal and apply all changes described in transactions to “in-place” locations (again)

(METADATA) JOURNALING



Breakdown of operations when writing changes in buffer cache

Example transaction “Txn 5”

Write-back and journaling phase:

1) Write data blocks in-place (i.e., to their final locations)
2) Write metadata blocks B0, D4, I2, IB1 into journal (order does not matter)
3) [Write Barrier]
4) Write transaction block T5 to mark “Txn 5” als valid
5) [Write Barrier]

Checkpointing phase:

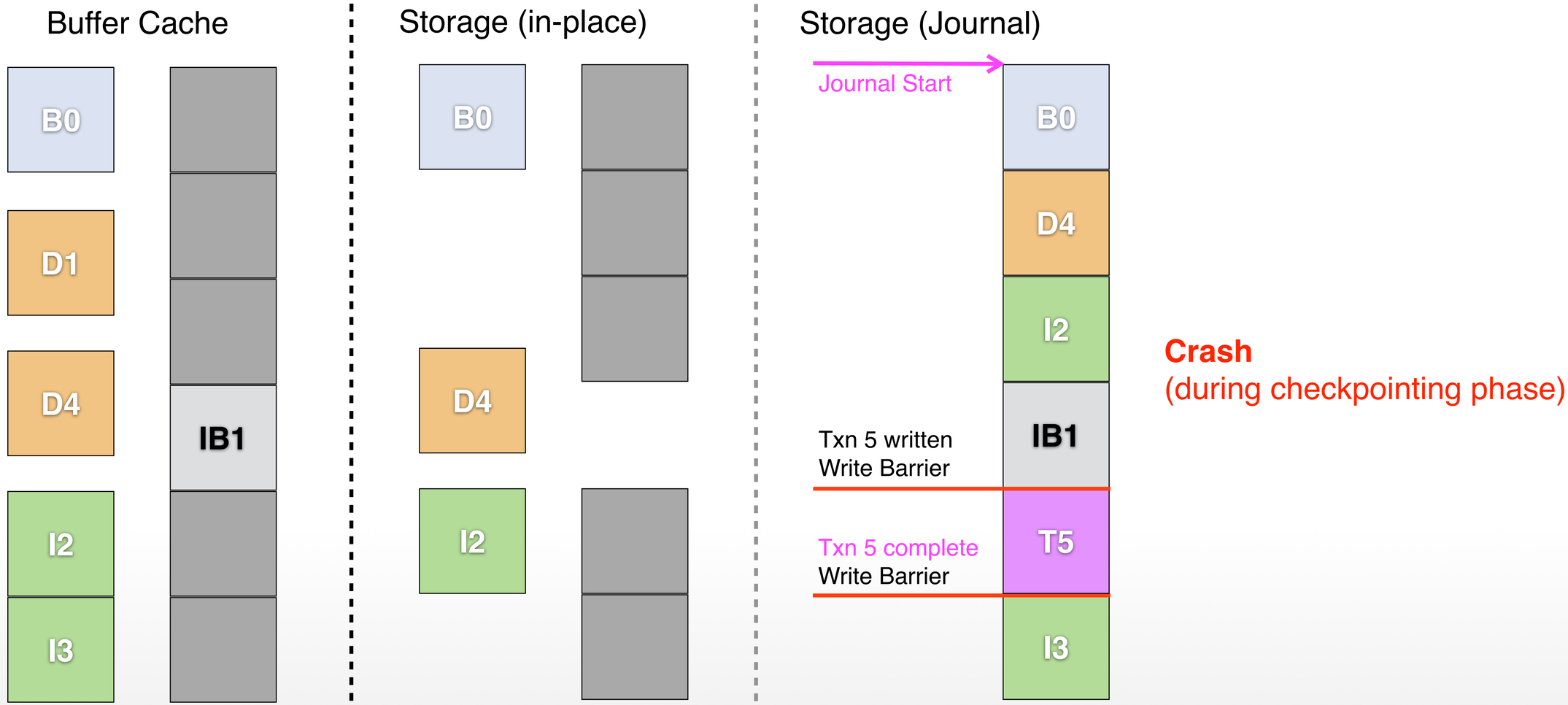
6) Write metadata blocks B0, D4, I2, IB1 a second time, now to their in-place locations
7) [Write Barrier]

Txn 5 done

8) Delete “Txn 5” from journal (by moving head pointer after Txn 5)

Inode block / Directory block / Indirect block / Bitmap block / Data block

CRASH WITH JOURNALING: CASE 1



Inode block / Directory block / Indirect block / Bitmap block / Data block

TRANSACTION REPLAY

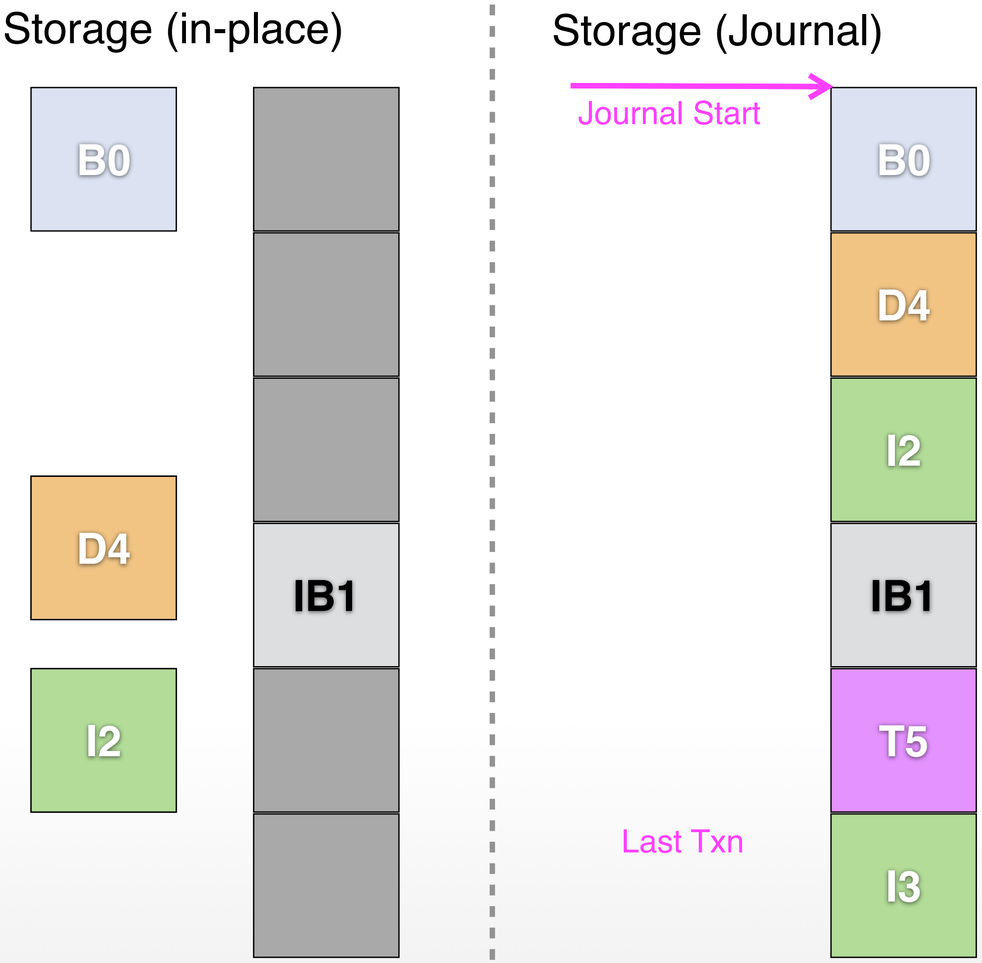
Case 1: Crash happened during checkpointing

All data blocks written, all metadata blocks in journal

Complete transaction is repeated

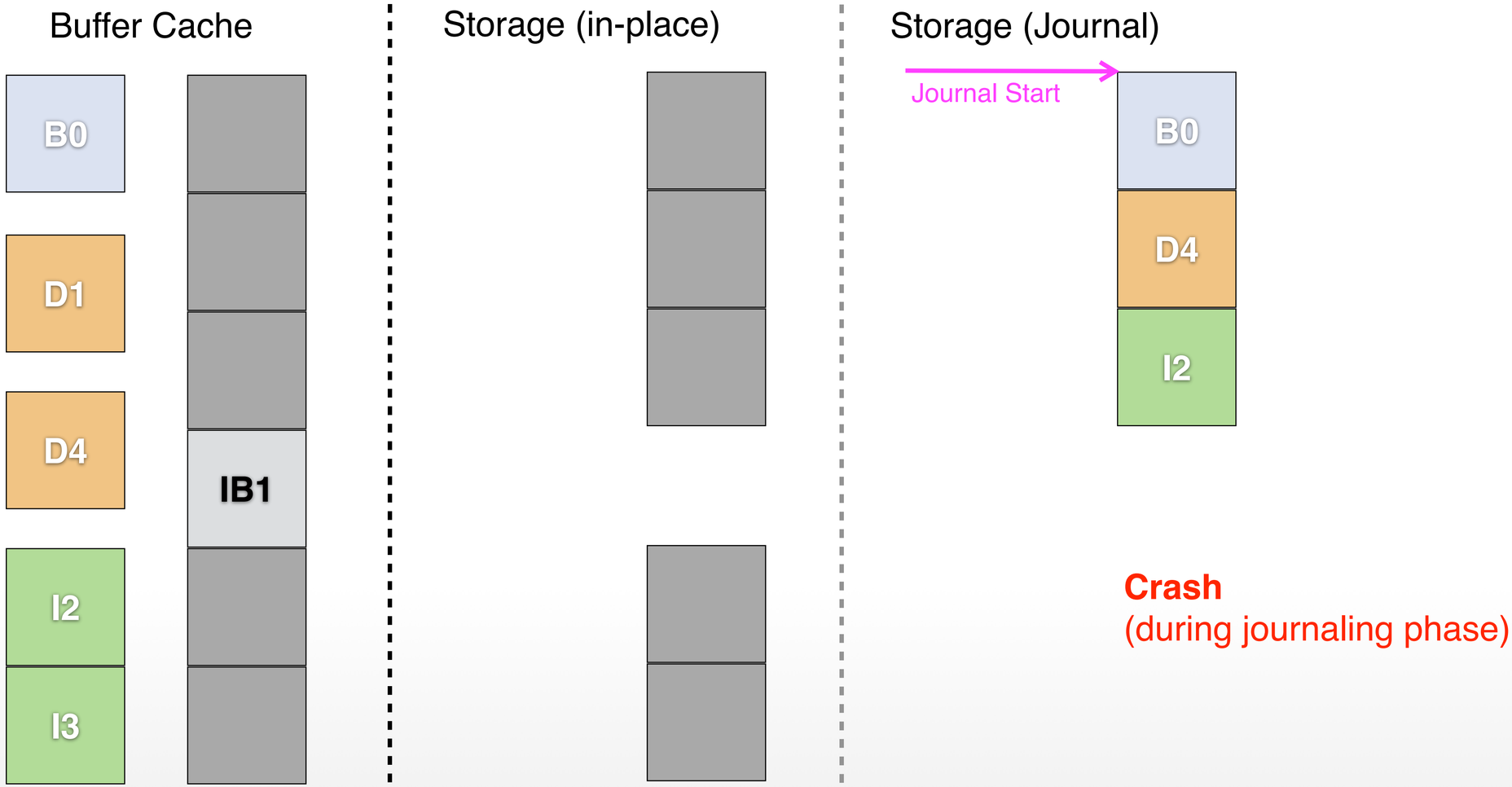
Metadata writes guaranteed to be complete at in-place addresses

In this example: new file exists!



Inode block / Directory block / Indirect block / Bitmap block / Data block

CRASH WITH JOURNALING: CASE 2



Inode block / Directory block / Indirect block / Bitmap block / Data block

TRANSACTION REPLAY

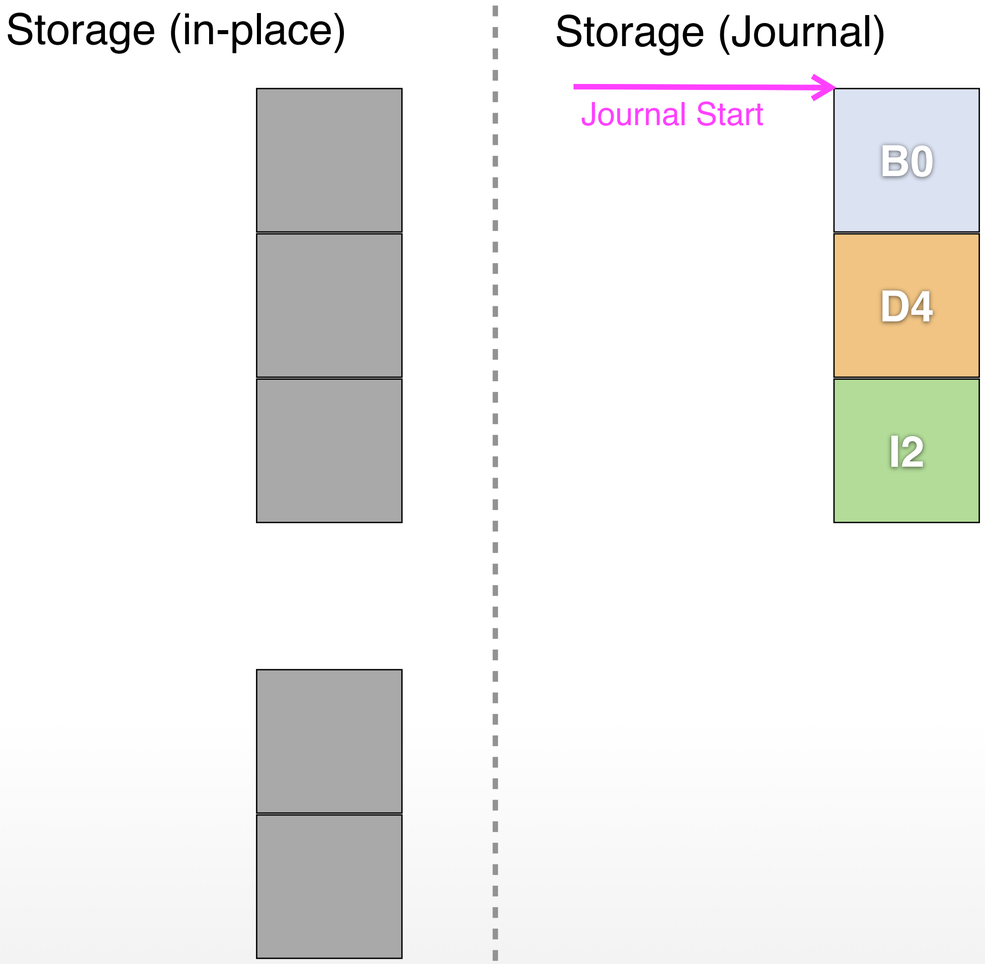
Case 2: Crash happened before completion of the journal transaction

Some data blocks written, some metadata blocks in journal

Transaction not complete

No replay is done

In this example: new file does not exist!



Inode block / Directory block / Indirect block / Bitmap block / Data block

PERFORMANCE

- **Write performance:**
  - Linear writes in journal (most efficient)
  - Checkpointing can be done with minimum number of seek operations (benefits from command queuing)
- **Minimization of write barriers:**
  1. Two write barriers to complete a transaction (which can be a very large “compound transaction”)
  2. One write barrier after checkpointing (can be combined with completion of next transaction)
- **Read performance:** unaffected by journaling

IMPLEMENTATION

- Journal format is **orthogonal** to rest of file system layout
  - Storage location for journal:
    - Fixed range of blocks (e.g., Reiserfs)
    - Hidden file, pre-allocated (e.g., Ext3/4, NTFS)
- **Granularity** of journal entries:
  - Complete metadata blocks (e.g., Ext3/4, Reiserfs)
    - Dependencies to blocks outside the transaction are forbidden
    - Problem similar to soft updates, but less complicated to solve
- Individual metadata updates (e.g., NTFS)

JOURNALIN MODES (EXT3)

- **Write-back journaling:**
  - Only metadata is logged in journal
  - Data blocks written “at some point”
  - Uninitialized file contents possible (but metadata consistent)
- **Ordered journaling [default setting]:**
  - Write all data blocks to “in-place”, then mark associated journal transaction as complete (one additional write barrier)
  - No uninitialized file contents (users want this!!!)
- **Data journaling:**
  - Data + metadata in journal (more expensive)

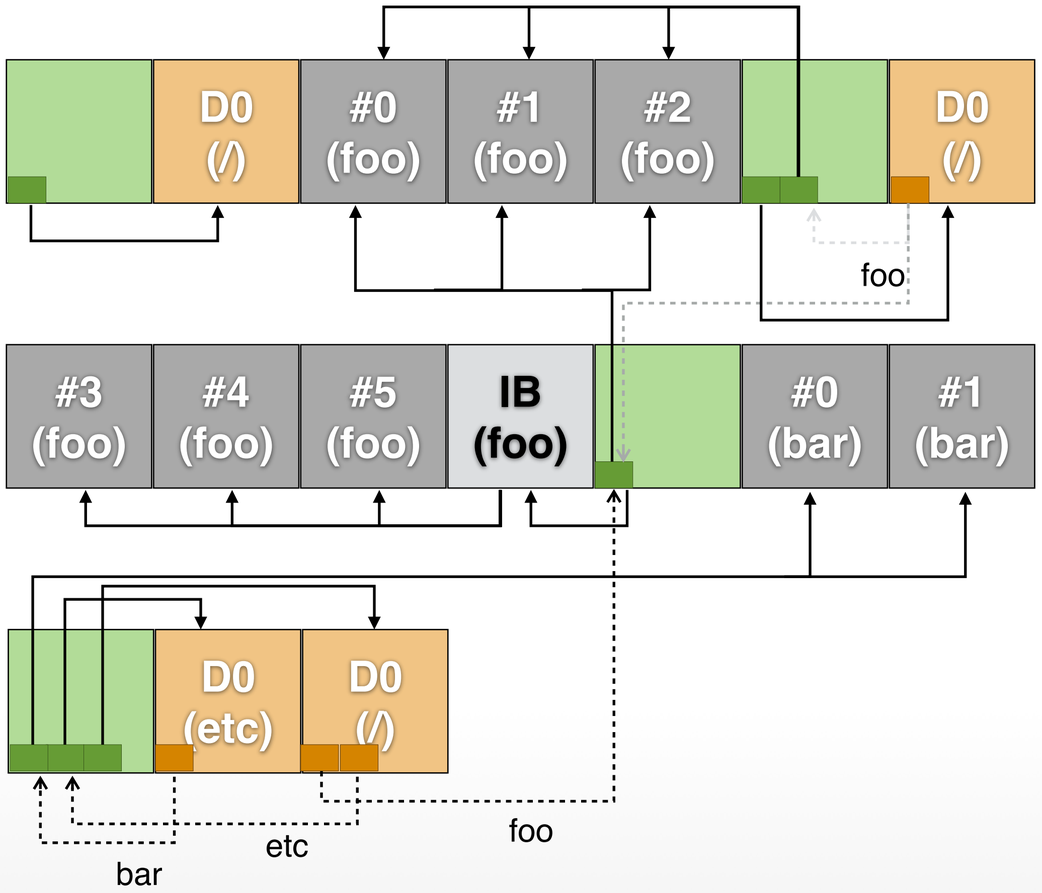
ROADMAP

- File system structures
- Inconsistencies after a crash
- Consistency mechanisms:
  - Synchronous writes
  - Soft updates
  - Journaling
  - Log-structured
  - Copy-on-write

LOG-STRUCTURED FILE SYSTEM

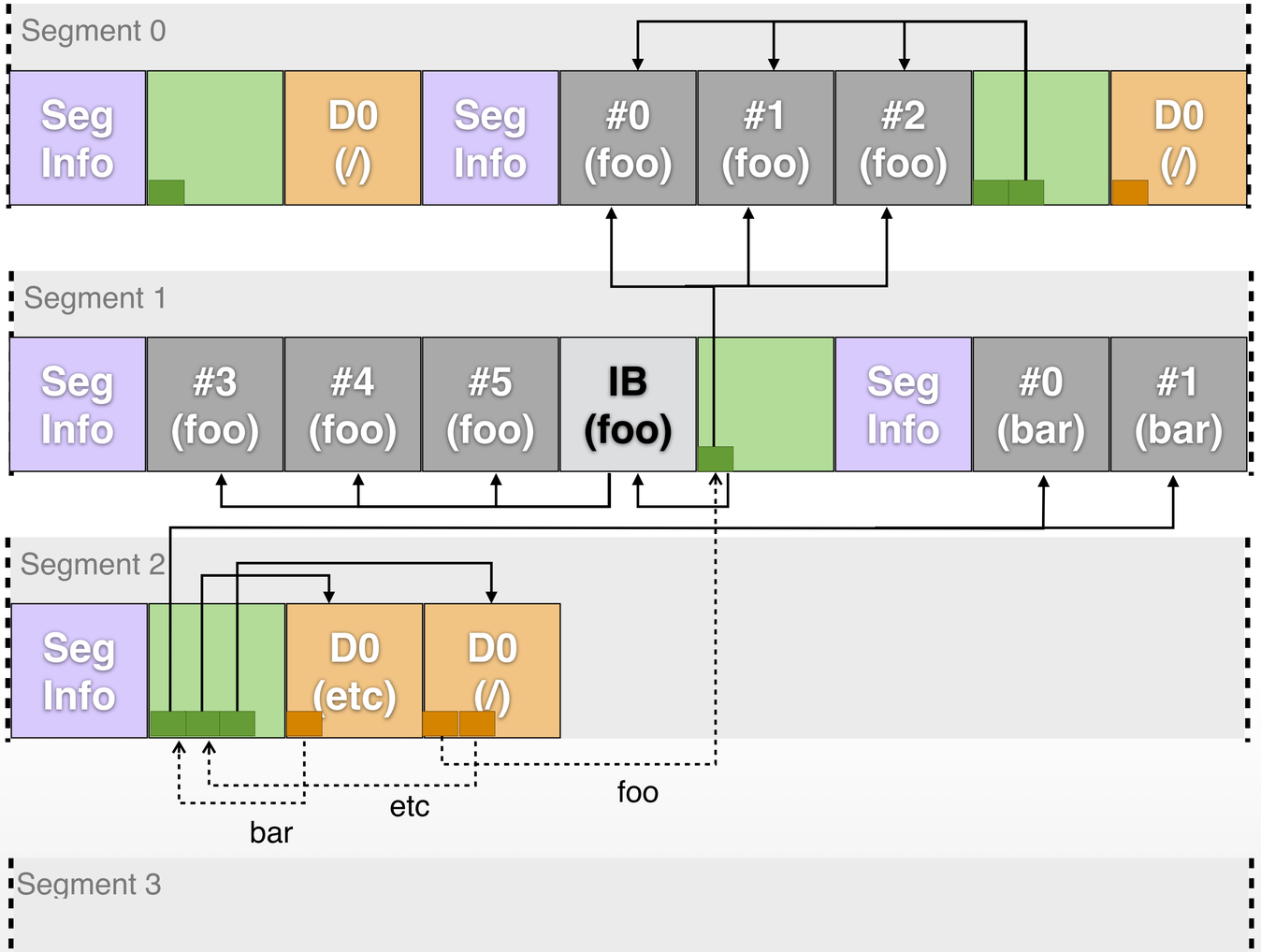
- **Idea:** treat entire file system as a log
  - Write data and metadata blocks linearly
  - No overwriting, append new block versions to end of log
  - Newest versions of all blocks contain latest file-system state
- **Additional features:**
  - Log contains consistent versions of old file-system state
  - Multiple **snapshots** of file-system state possible

LOG-STRUCTURED FILE SYSTEM



Inode block / Directory block / Indirect block / Data block

LOG SEGMENTATION



Inode block / Directory block / Indirect block / Data block / Summary block

LOG SEGMENTATION

- Block address space portioned into **segments**
- Segments contain both data and metadata blocks
- **Summary blocks** describe block types:
  - File / directory blocks: (Inode,Block#)
  - Inode blocks: new / modified inodes
  - **Inode-map** blocks: pointers to inodes in log
- Inode map **decouples** inode pointers in directory entries from physical location in log (prevents trickle-up effect up to root directory)

CHECKPOINTS

- All data and metadata in log …
  But: full scan of entire log too expensive
- **Better:** regularly checkpoint current state of metadata in compact form
- **2 checkpoint areas** at fixed locations:
  - **Pointers** to all blocks (in log) that contain up-to-date and consistent versions of inode map + segment usage table
  - **Timestamp** + **pointer** to last written segment
  - **Checksum** to validate consistency and completeness of checkpoint area

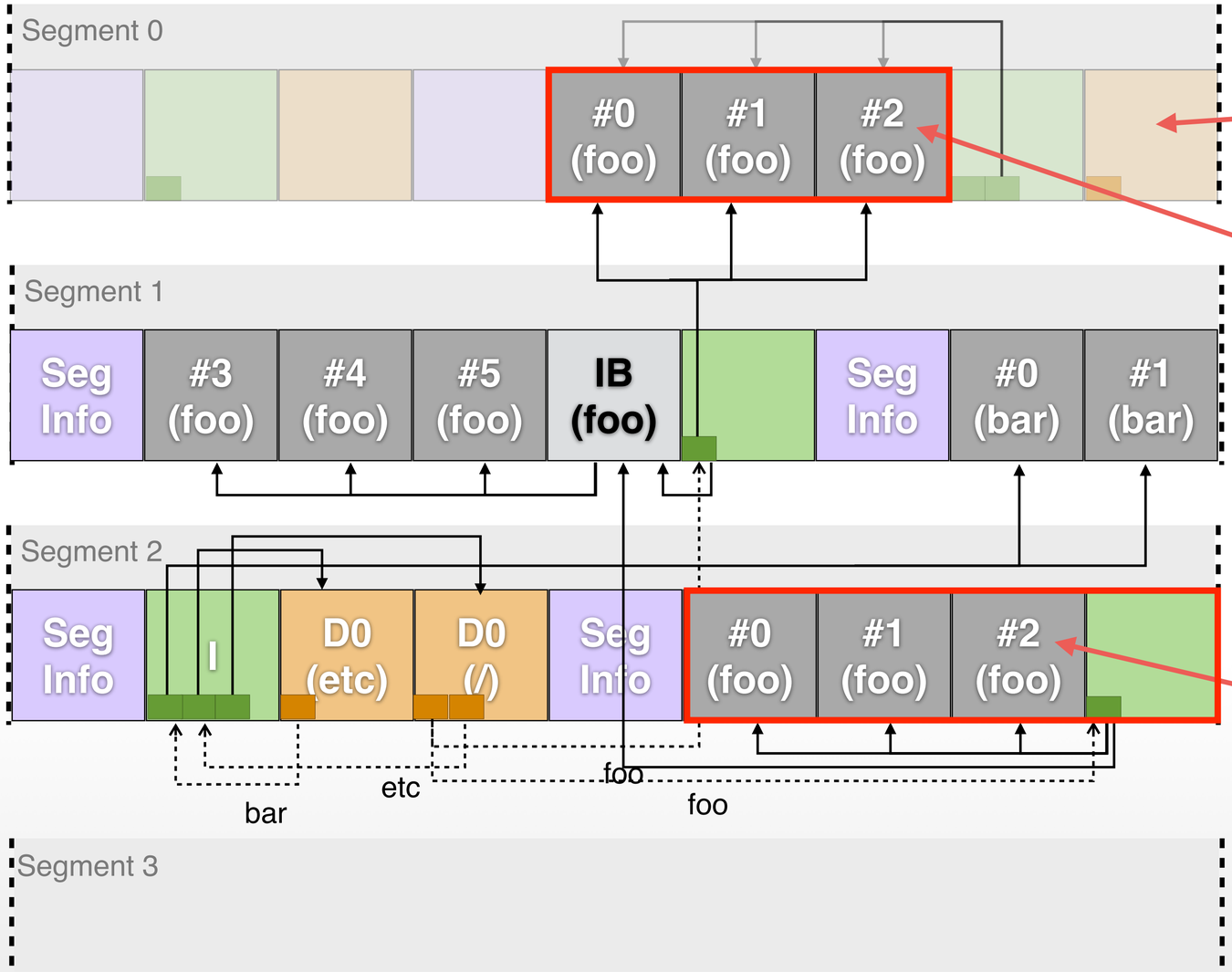
MOUNTING AND CRASH RECOVERY

- **Replay from last checkpoint:**
  1. Choose most recent + consistent checkpoint area
  2. Reinitialize inode map in kernel memory
  3. Determine position of last segment written before checkpoint
  4. Roll-forward all changes that have been written to the log after the last checkpoints: **update** in-kernel data structures (inode map, segment usage table, etc.) by scanning all post-checkpoint file-system state found in the log
  5. Latest consistent file system has been recovered

THE LIMITS TO GROWTH

- **Problem:** Log cannot grow indefinitely
- **Solution 1:** free old blocks, reintegrate them into log
  - (+) Simple and fast free operartion
  - (-) Fragmentation is introduced and becomes worse
- **Solution 2:** free complete segements
  - (+) Fragmentation irrelevant (because segments are big)
  - (-) Live blocks in segment must be copied into new segment
- In practice: only **Solution 2** (LFS [3] and others)

GARBAGE COLLECTION



Obsolete versions of blocks

Live versions of blocks in a segment that could be freed, if the live blocks were moved to another segment

New copies of live blocks + updated inode block pointing to these blocks appended to the log

GARBAGE COLLECTION

- **Cleaner process** searches for partially live segments in background
- Starts and stops based on thresholds (low / high watermark of free segments)
- There is no block-allocation table or free list; how to identify obsolete blocks?
  1. **Find pointer** to block in inode or indirect block
  2. **No valid pointer found?** → block is **obsolete**
- More details in [3]

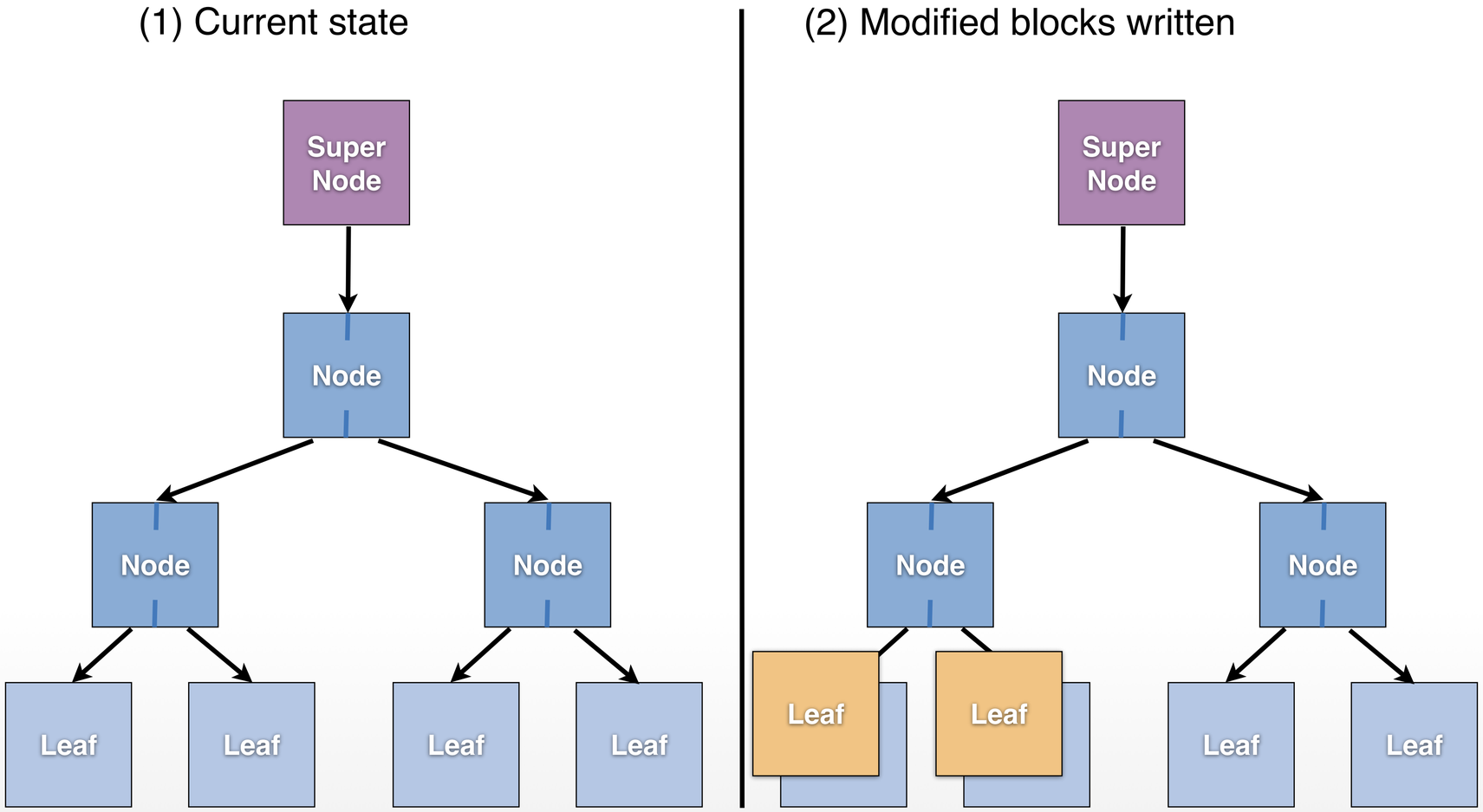
ROADMAP

- File system structures
- Inconsistencies after a crash
- Consistency mechanisms:
  - Synchronous writes
  - Soft updates
  - Journaling
  - Log-structured
  - Copy-on-write

COPY-ON-WRITE

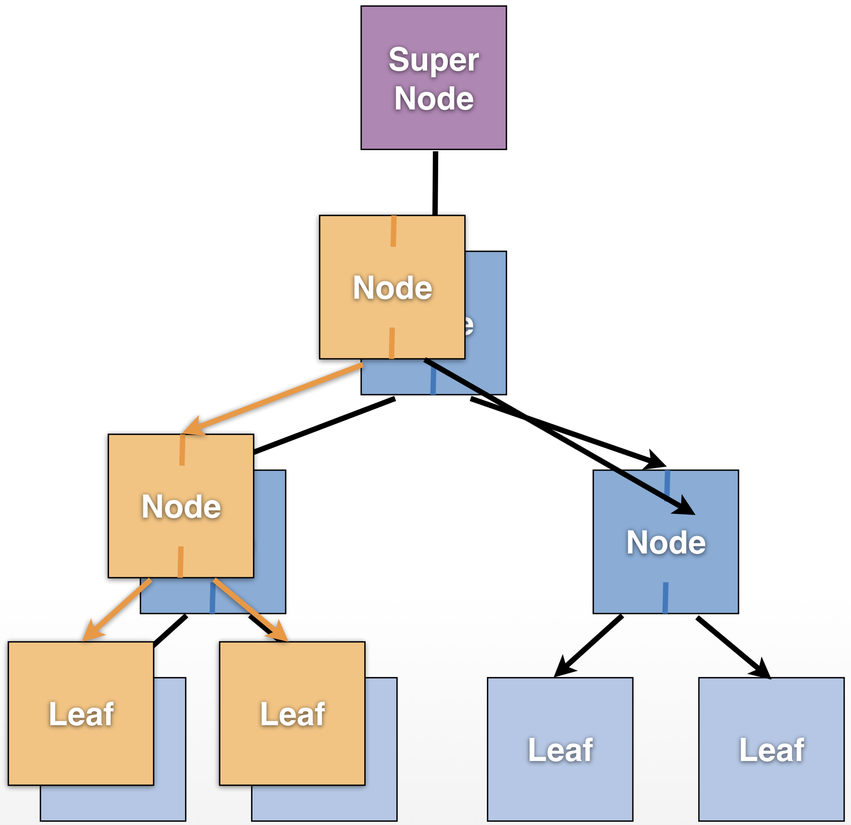
- **Basic idea** similar to log-structured file systems:
  Never overwrite!
- Entire file system is a B+ tree (or hierarchy of B+ trees)
- Changes to file system structures:
  - **Copy-on-write:** write a modified version of a **tree node** (i.e., block) to a free position
  - Update pointers in parent nodes afterwards
    → copy-on-write updates of parent nodes, up to the root
- Update to root pointer will switch **atomically** to a new file system state

COPY-ON-WRITE

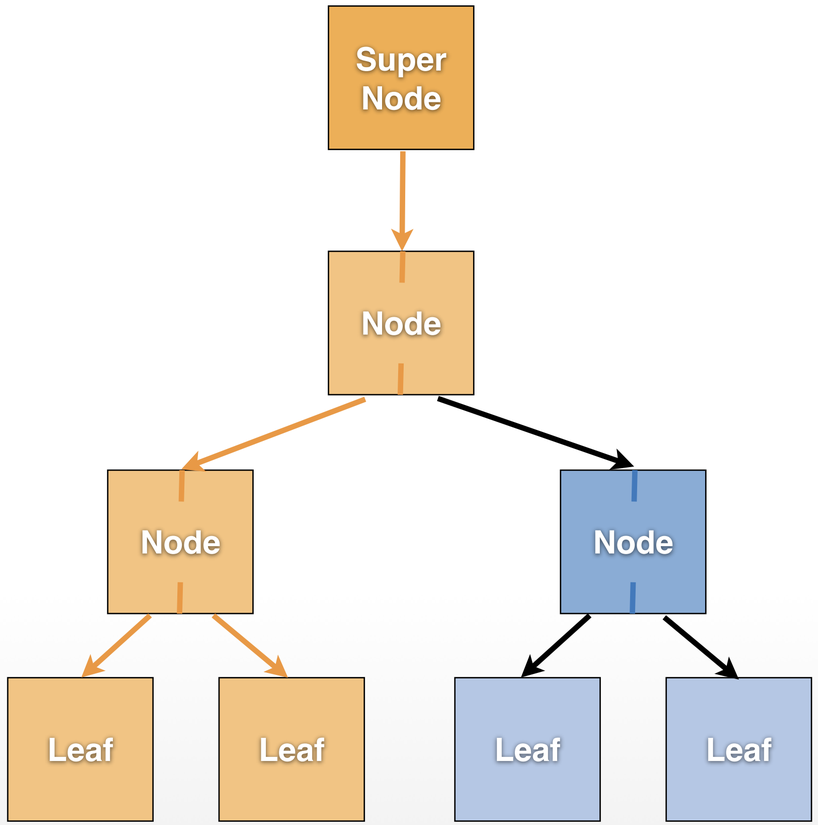


COPY-ON-WRITE

(3) Parent nodes updated



(4) Root node updated, old versions freed



IS THIS ALL STILL RELEVANT?

- Journaling approach: *Ext3/Ext4, XFS, NTFS, …*
- Log-structured approach:
  - Linux: *JFFS2, YAFFS, NILFS2, F2FS*
  - In SSDs: Flash Translation Layer (FTL)
- Copy-on-write approach:
  - BSD / Linux: *ZFS, BTRFS*
  - All Apple OSes: *APFS*
  - Windows 8+: *ReFS* (now restricted to server editions)
- Synchronous writes: persistent memory file systems

COMPARISON: FS/HDD/SSD

Strategie	Harddisk	Solid State Disk
Synchronous	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: Read: **R**, Write: **W**

Performance: very bad **-- - o + ++** very good

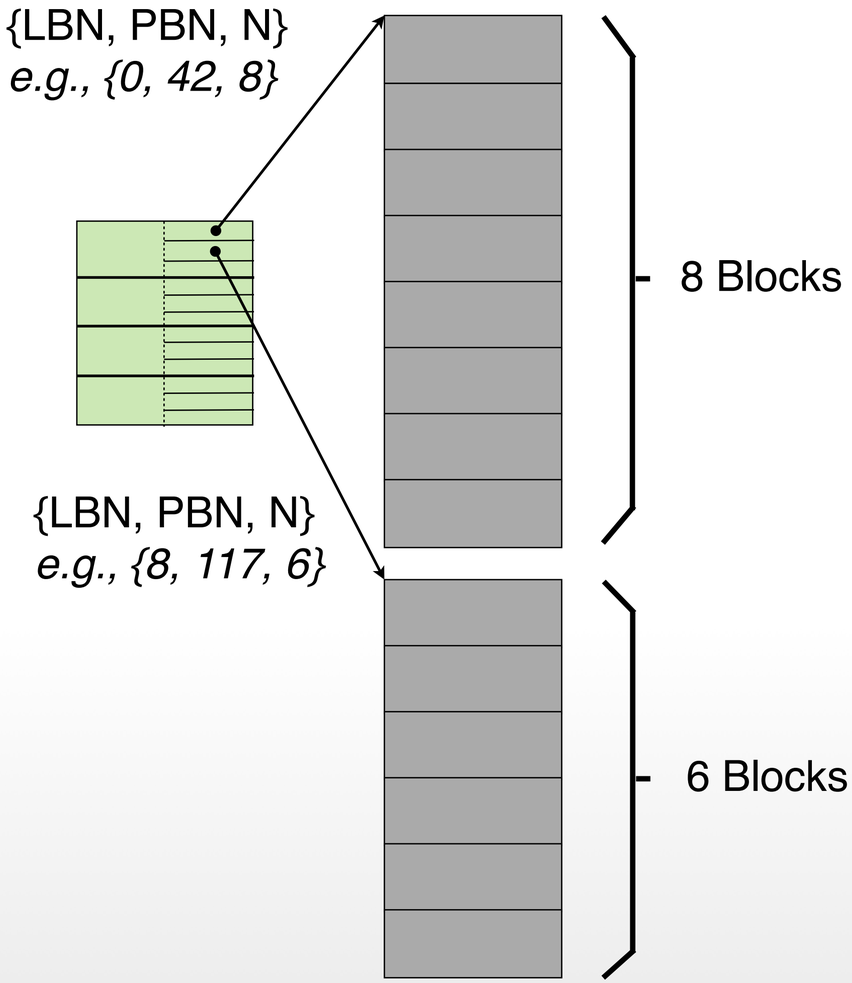
ADDENDUM: FILE SYSTEM STRUCTURES

OPTIMIZATIONS

- **Very small files** (smaller than block size):
  - **Inline data:** storage of fewer bytes directly in inode (e.g. new in Ext4)
  - **Tail packing:** many file remnants in one block
- **Very large files:**
  - Problem: unnecessarily large number of block pointers
    - Pointers to directly consecutive blocks
    - High storage requirements, indirect blocks
- Solution: **Extents**

EXTENTS

- **Extents:** sequence of contiguous blocks
  - **LBN:** logical block number within file
  - **PBN:** physical block number of first block
  - **N:** number of blocks
- Advantage: few extent descriptors instead of many block pointers



EXTENTS IN EXT4

- Example **Ext4:**
  - Maximum of 128 MB per extent
  - 4 extend descriptors per inode
  - Very large / heavily fragmented files:
    - Multi-level extent trees, if more than four leaves are needed
    - Similar to indirect blocks, but nodes are extent descriptors instead of pointers to individual blocks
- Most modern file systems use extents

SOURCES

[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