Advanced File Systems

CS 140 – Nov 4, 2016

Ali Jose Mashtizadeh

Outline

- **FFS Review and Details**
- Crash Recoverability
- Soft Updates
- Journaling
- Copy-on-Write: LFS/WAFL/ZFS

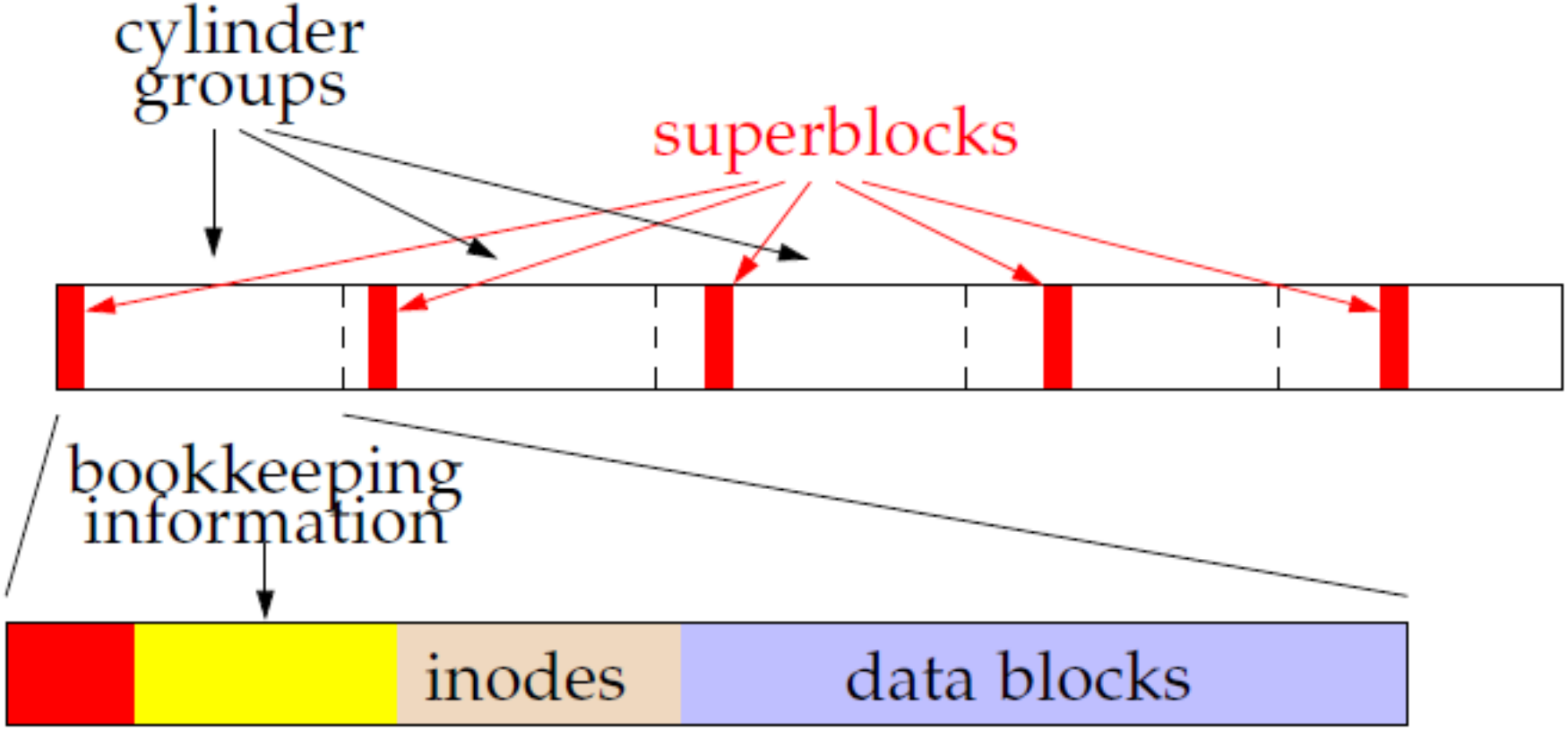
Review: Improvements to UNIX FS

- Problems with original UNIX FS
  - 512 B blocks
  - Free blocks in linked list
  - All inodes at the beginning of the disk
- UNIX Performance Issues
  - Transfers 512 B per disk IO
  - Fragmentation leads to 512 B/average seek
  - Inodes far from directory and file data
  - Files within directory scattered everywhere
- Useability Issues
  - 14 character file names
  - not crash proof

Review: Fast File System [McKusic]

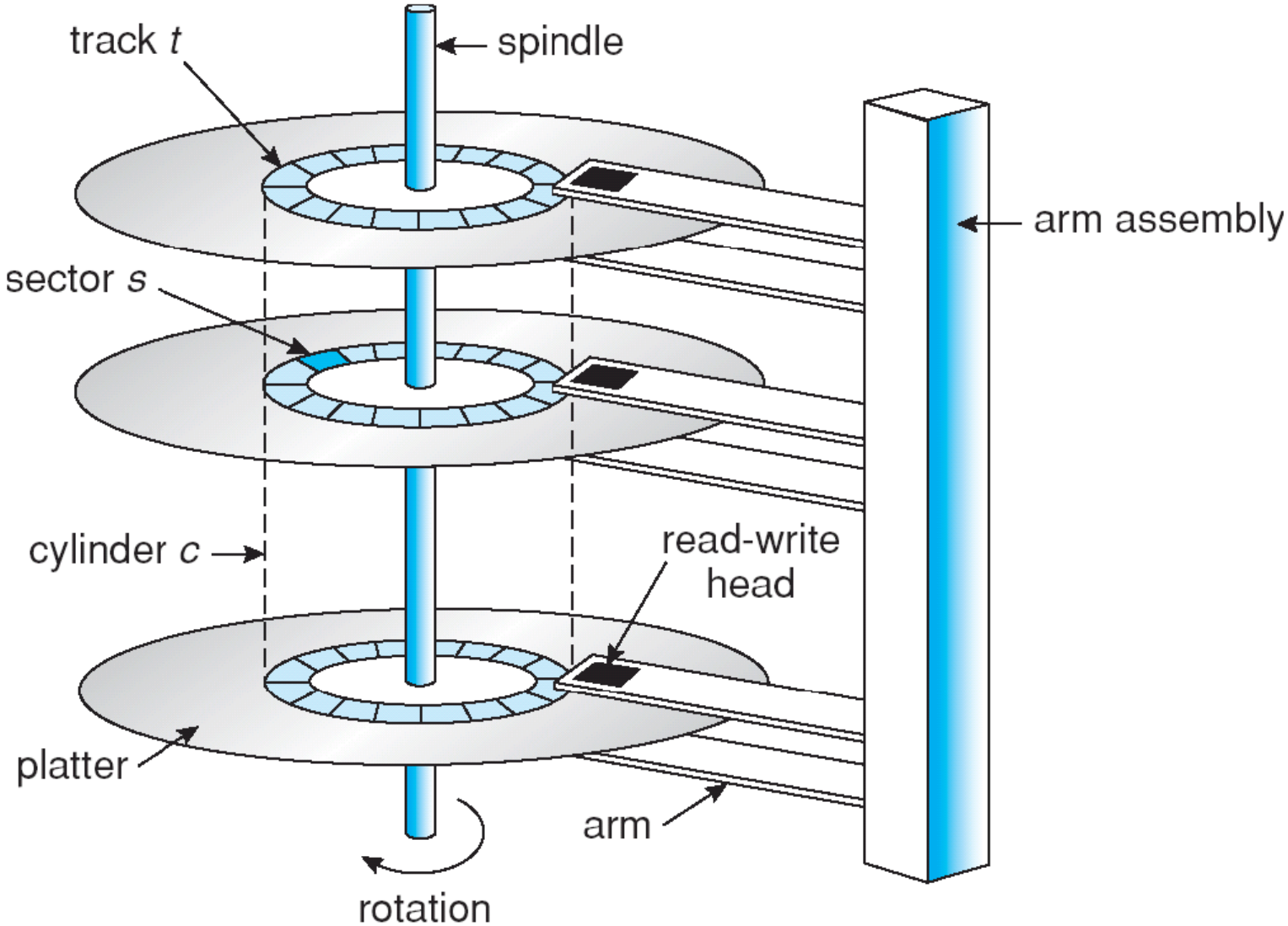
- Variable block size (at least 4 KiB)
  - Fragment technique reduced wasted space
- Cylinder groups spread inodes around disk
- Bitmap for fast allocation
- FS reserves space to improve allocation
  - Tunable parameter default 10%
  - Reduces fragmentation
- Usability improvements:
  - 255 character file names
  - Atomic rename system call
  - Symbolic links

Review: FFS Disk Layout



- Each cylinder group has its own:
  - Superblock
  - Cylinder Block: Bookkeeping information
  - Inodes, Data/Directory blocks

Cylinders, tracks, and sectors



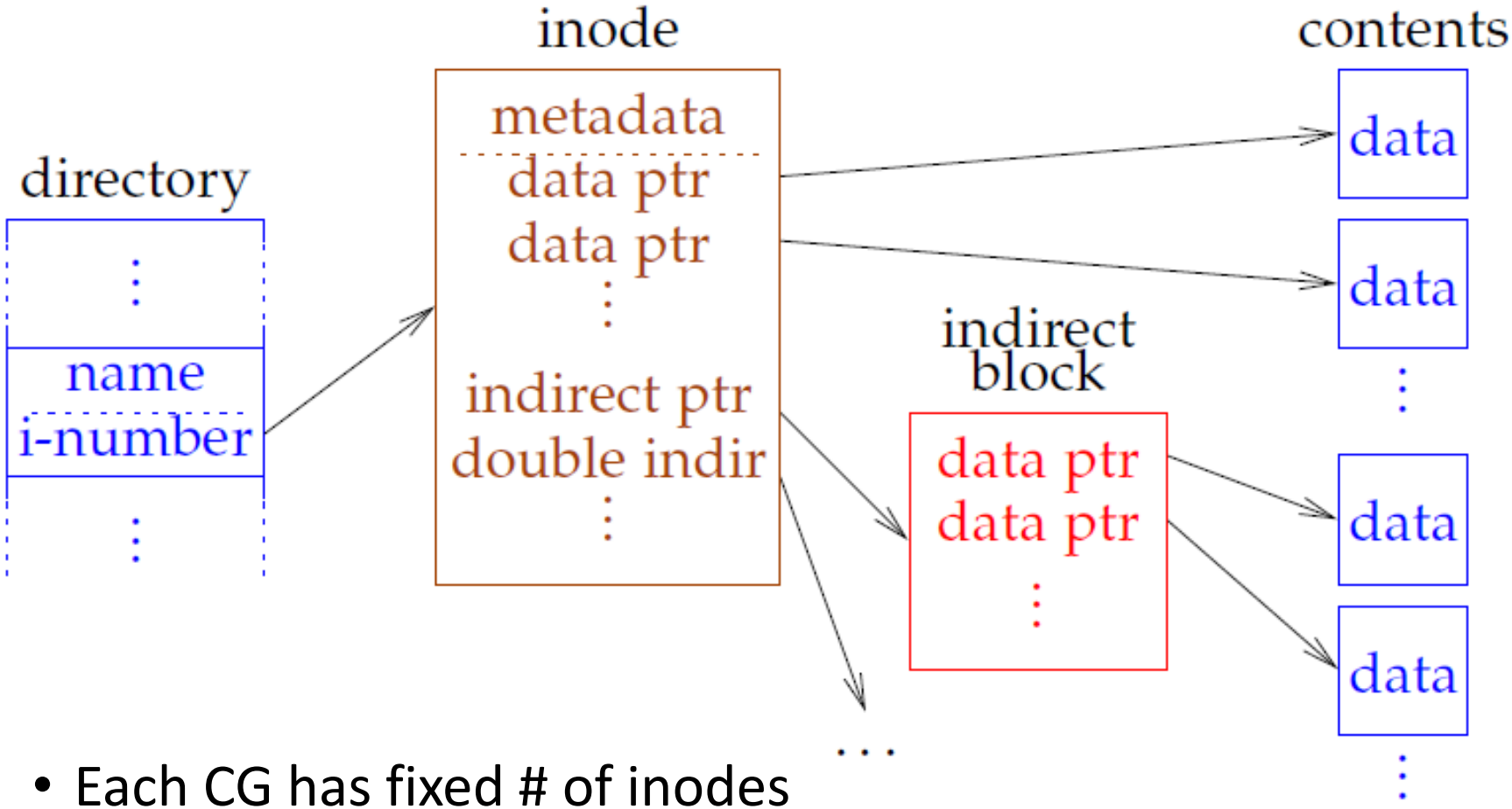
Superblock

- Contains file system parameters
  - Disk characteristics, block size, Cyl. Group info
  - Information to locate inodes, free bitmap, and root dir.
- Replicated once per cylinder group
  - At different offsets to span multiple platters
  - Contains magic number 0x00011954 to find replica's (McKusick's birthday)
- Contains non-replicated information (FS summary)
  - Number of blocks, fragments, inodes, directories
  - Flag stating if the file system was cleanly unmounted

Bookkeeping information

- Block map
  - Bitmap of available fragments
  - Used for allocating new blocks/fragments
- Summary info within CG (Cylinder Group Block)
  - # of free inodes, blocks/frags, files, directories
  - Used when picking which cyl. group to allocate into
- # of free blocks by rotational position (8 positions)
  - Reasonable when disks were accessed with CHS
  - OS could use this to minimize rotational delay

Inodes and Data blocks



- Each CG has fixed # of inodes
- Each inode maps offset to disk block for one file
- Inode contains metadata

On-Disk Inode

```
struct ufs1_dinode {
        u_int16_t di_mode;              /* 0: IFMT, permissions; see below. */
        int16_t di_nlink;               /* 2: File link count. */
        uint32_t di_freelink;           /* 4: SUJ: Next unlinked inode. */
        u_int64_t di_size;              /* 8: File byte count. */
        int32_t di_atime;               /* 16: Last access time. */
        int32_t di_atimensec;           /* 20: Last access time. */
        int32_t di_mtime;               /* 24: Last modified time. */
        int32_t di_mtimensec;           /* 28: Last modified time. */
        int32_t di_ctime;               /* 32: Last inode change time. */
        int32_t di_ctimensec;           /* 36: Last inode change time. */
        ufs1_daddr_t di_db[NDADDR];     /* 40: Direct disk blocks. */
        ufs1_daddr_t di_ib[NIADDR];     /* 88: Indirect disk blocks. */
        u_int32_t di_flags;             /* 100: Status flags (chflags). */
        u_int32_t di_blocks;            /* 104: Blocks actually held. */
        u_int32_t di_gen;               /* 108: Generation number. */
        u_int32_t di_uid;               /* 112: File owner. */
        u_int32_t di_gid;               /* 116: File group. */
        u_int64_t di_modrev;            /* 120: i_modrev for NFSv4 */
};
```

On-Disk Inode: POSIX Permissions

```
struct ufs1_dinode {
    u_int16_t di_mode;          /* 0: IFMT, permissions; see below. */
    int16_t di_nlink;           /* 2: File link count. */
    uint32_t di_freelink;       /* 4: SUJ: Next unlinked inode. */
    u_int64_t di_size;          /* 8: File byte count. */
    int32_t di_atime;           /* 16: Last access time. */
    int32_t di_atimensec;       /* 20: Last access time. */
    int32_t di_mtime;           /* 24: Last modified time. */
    int32_t di_mtimensec;       /* 28: Last modified time. */
    int32_t di_ctime;           /* 32: Last inode change time. */
    int32_t di_ctimensec;       /* 36: Last inode change time. */
    ufs1_daddr_t di_db[NDADDR]; /* 40: Direct disk blocks. */
    ufs1_daddr_t di_ib[NIADDR]; /* 88: Indirect disk blocks. */
    u_int32_t di_flags;         /* 100: Status flags (chflags). */
    u_int32_t di_blocks;        /* 104: Blocks actually held. */
    u_int32_t di_gen;           /* 108: Generation number. */
    u_int32_t di_uid;           /* 112: File owner. */
    u_int32_t di_gid;           /* 116: File group. */
    u_int64_t di_modrev;        /* 120: i_modrev for NFSv4 */
};
```

On-Disk Inode: Hard Link Count

```
struct ufs1_dinode {
        u_int16_t di_mode;              /* 0: IFMT, permissions; see below. */
        int16_t di_nlink;               /* 2: File link count. */
        uint32_t di_freelink;           /* 4: SUJ: Next unlinked inode. */
        u_int64_t di_size;              /* 8: File byte count. */
        int32_t di_atime;               /* 16: Last access time. */
        int32_t di_atimensec;           /* 20: Last access time. */
        int32_t di_mtime;               /* 24: Last modified time. */
        int32_t di_mtimensec;           /* 28: Last modified time. */
        int32_t di_ctime;               /* 32: Last inode change time. */
        int32_t di_ctimensec;           /* 36: Last inode change time. */
        ufs1_daddr_t di_db[NDADDR];     /* 40: Direct disk blocks. */
        ufs1_daddr_t di_ib[NIADDR];     /* 88: Indirect disk blocks. */
        u_int32_t di_flags;             /* 100: Status flags (chflags). */
        u_int32_t di_blocks;            /* 104: Blocks actually held. */
        u_int32_t di_gen;               /* 108: Generation number. */
        u_int32_t di_uid;               /* 112: File owner. */
        u_int32_t di_gid;               /* 116: File group. */
        u_int64_t di_modrev;            /* 120: i_modrev for NFSv4 */
};
```

On-Disk Inode: Block Pointers

```
struct ufs1_dinode {
        u_int16_t di_mode;              /* 0: IFMT, permissions; see below. */
        int16_t di_nlink;               /* 2: File link count. */
        uint32_t di_freelink;           /* 4: SUJ: Next unlinked inode. */
        u_int64_t di_size;              /* 8: File byte count. */
        int32_t di_atime;               /* 16: Last access time. */
        int32_t di_atimensec;           /* 20: Last access time. */
        int32_t di_mtime;               /* 24: Last modified time. */
        int32_t di_mtimensec;           /* 28: Last modified time. */
        int32_t di_ctime;               /* 32: Last inode change time. */
        int32_t di_ctimensec;           /* 36: Last inode change time. */
        ufs1_daddr_t di_db[NDADDR];     /* 40: Direct disk blocks. */
        ufs1_daddr_t di_ib[NIADDR];     /* 88: Indirect disk blocks. */
        u_int32_t di_flags;             /* 100: Status flags (chflags). */
        u_int32_t di_blocks;            /* 104: Blocks actually held. */
        u_int32_t di_gen;               /* 108: Generation number. */
        u_int32_t di_uid;               /* 112: File owner. */
        u_int32_t di_gid;               /* 116: File group. */
        u_int64_t di_modrev;            /* 120: i_modrev for NFSv4 */
};
```

Inode Allocation

- Every file/directory requires an inode
- New file: Place inode/data in same CG as directory
- New directory: Use a different CG from parent
  - Select a CG with greater than average # free inodes
  - Choose a CG with smallest # of directories
- Within CG, inodes allocated randomly
  - All inodes close together as CG doesn’t have many
  - All inodes can be read and cached with few IOs

Fragment allocation

- Allocate space when user grows a file
- Last block should be a fragment if not full-size
  - If growth is larger than a fragment move to a full block
- If no free fragments exist break full block
- Problem: Slow for many small writes
  - May have to keep moving the end of the file around
- Solution: new stat struct field st_blksize
  - Tells applications and stdio library to buffer this size

Block allocation

- Optimize for sequential access
  - #1: Use block after the end of file
  - #2: Use block in same cylinder group
  - #3: Use quadratic hashing to choose next CG
  - #4: Choose any CG
- Problem: don’t want one file to fill up whole CG
  - Otherwise other inodes will not be near data
- Solution: Break big files over many CGs
  - Large extents in each CGs, so seeks are amortized
  - Extent transfer time >> seek time

Directories

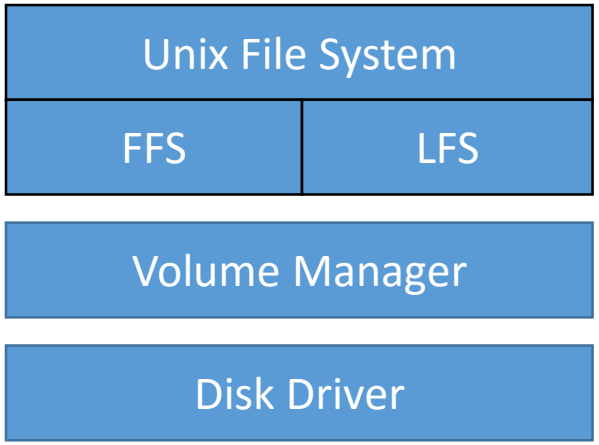
- Same as a file, but Inode marks as a directory
- Contents considered as 512-byte chunks
  - Disks only guarantee atomic sector updates
- Each chunk has `direct` structure with:
  - 32-bit i-number
  - 16-bit size of directory entry
  - 8-bit file type
  - 8-bit length of file name
- Coalesce when deleting
- Periodically compact directory chunks
  - Never move across chunks

Updating FFS for the 90s

- No longer wanted to assume rotational delay
  - Disk caches usually reduce rotational delay effects
  - Data contiguously allocated
- Solution: Cluster writes
  - File system delays writes
  - Accumulates data into 64 KiB clusters to write at once
- Cluster allocation similar to fragment/blocks
- Online Defragmentation
  - Portions of files can be copied to future read cost
- Better Crash Recovery
  - FreeBSD: Normal fsck, GJournal, SU, SU+J

FFS Implementation

- Separates
  - File/Directory abstraction (UFS)
  - Disk Layout (FFS)
- Log File System (LFS) [Mendel]
  - Log structured file system layout for BSD
- Disk Layout
  - Maps i-number to inode
  - Manages free space
  - IO interface given inode



Outline

- Overview
- FFS Review and Details
- **Crash Recoverability**
- Soft Updates
- Journaling
- LFS/WAFL

Fixing Corruptions

- File System Check (fsck) run after crash
  - Hard disk guarentees per-sector atomic updates
  - File system operates on blocks (~8 sectors)
- Summary info usually bad
  - Recount inodes, blocks, fragments, directories
- System may have corrupt inodes
  - Inodes may have bad fields, extended attributes corrupt
  - Inodes < 512 B
- Allocated blocks may be missing references
- Directories may be corrupt
  - Holes in directory
  - File names corrupt/not unique/corrupt inode number
  - All directories must be reachable

Ensure Recoverability

- Goal: Ensure fsck can recover the file system
- Example: suppose we asynchronously write data
- Appending:
  - Inode points to a corrupt indirect block
  - File grown, but new data not present
- Delete/truncate + Append to another file:
  - New file may reuse old block
  - Old inode not yet written to disk
  - Cross allocation!

Performance & Consistency

- We need to guarantee some ordering of updates
  - Write new inode to disk before directory entry
  - Remove directory name before deallocation
  - Write cleared inode to disk before updating free bitmap
- Requires many metadata writes to be synchronous
  - Ensures easy recovery
  - Hurts performance
  - Cannot always batch updates
- Performance:
  - Extracting tar files easily 10-20x slower
  - fsck: very slow leading to more downtime!

Outline

- Overview
- FFS Review and Details
- Crash Recoverability
- **Soft Updates**
- Journaling
- LFS/WAFL

Ordered Updates

- Follow 3 rules for ordering updates [Ganger]
  - Never write pointer before initializing the structure pointed to
  - Never reuse a resource before nullifying all pointers to it
  - Never clear last pointer to live resource before setting new one
- File system will be recoverable
- Might leak disk space, but file system correct
- Goal: scavenge in background for missing resources

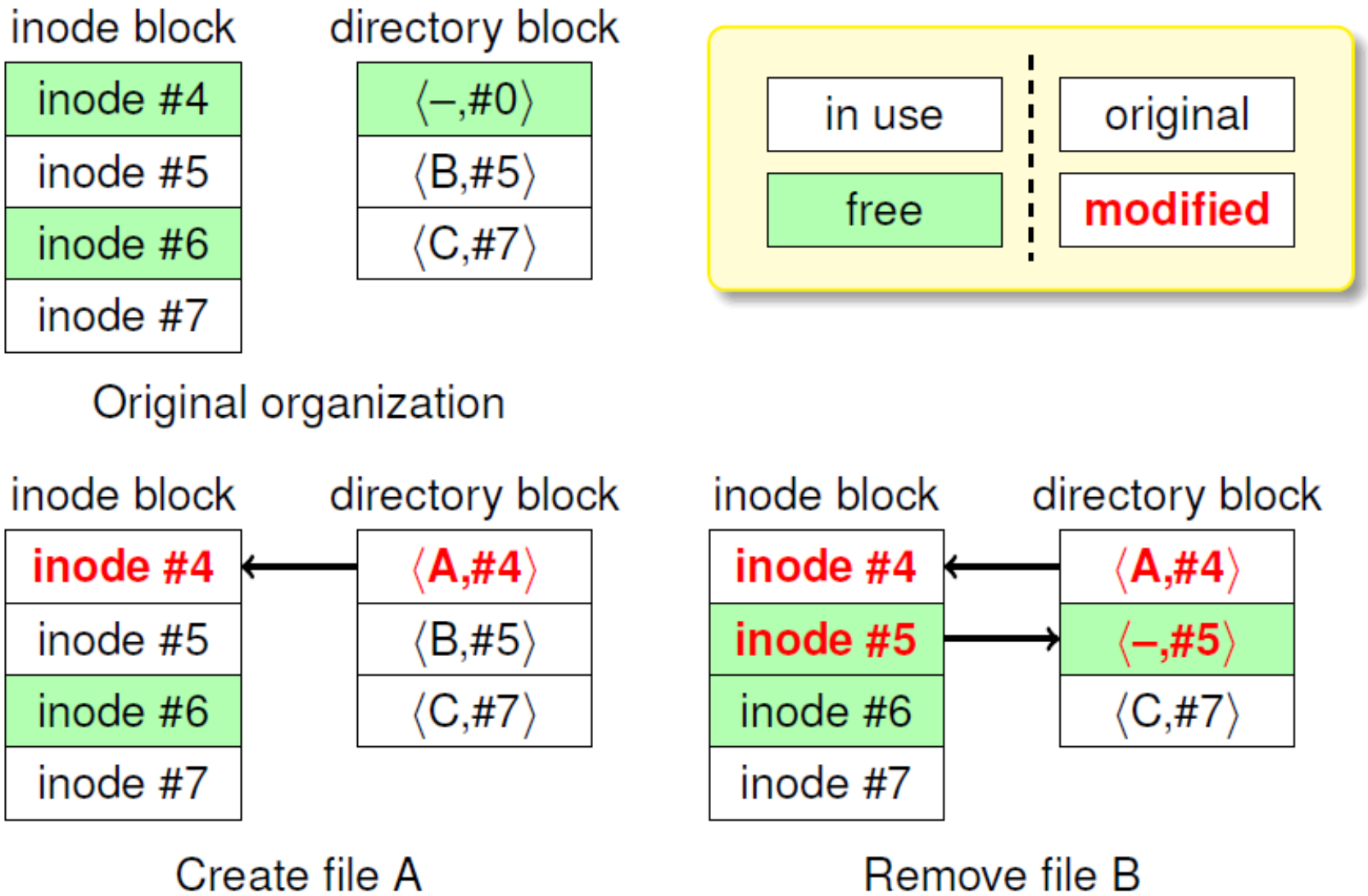
Dependencies

- Example: Creating a file A
  - Block X contains file A's inode
  - Block Y contains directory block references A
- We say "Y depends on X"
  - Y cannot be written before X is written
  - X is the *dependee*, Y is the *depender*
- We can hold up writes, but must preserve order

Cyclic Dependencies

- Suppose you create A, unlink B
  - Both have same directory & inode
- Cannot write directory until A is initialized
  - Otherwise: Directory will point to bogus inode
  - So A might be associated with the wrong data
- Cannot write inode until B's directory entry cleared
  - Otherwise B could end up with a too small link count
  - File could be deleted while links exist
- Otherwise, fsck has to be slower!
  - Check every directory entry and inode link counts
  - Requires more memory

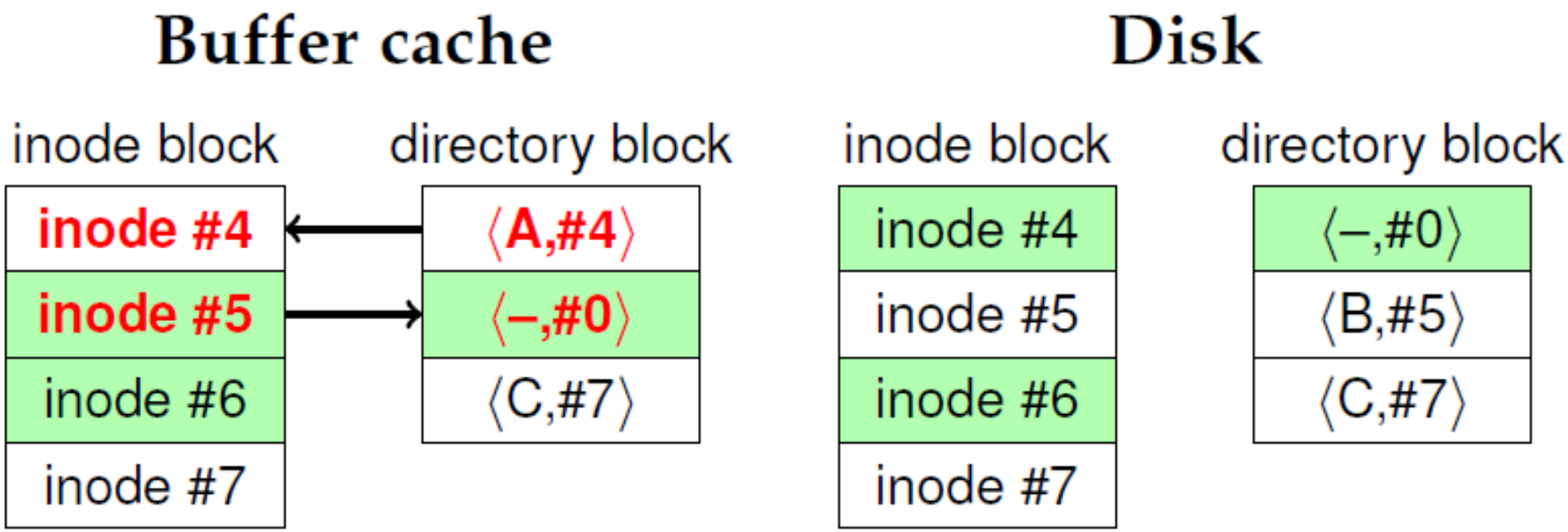
Cyclic Dependencies Illustrated



Soft Updates [Ganger]

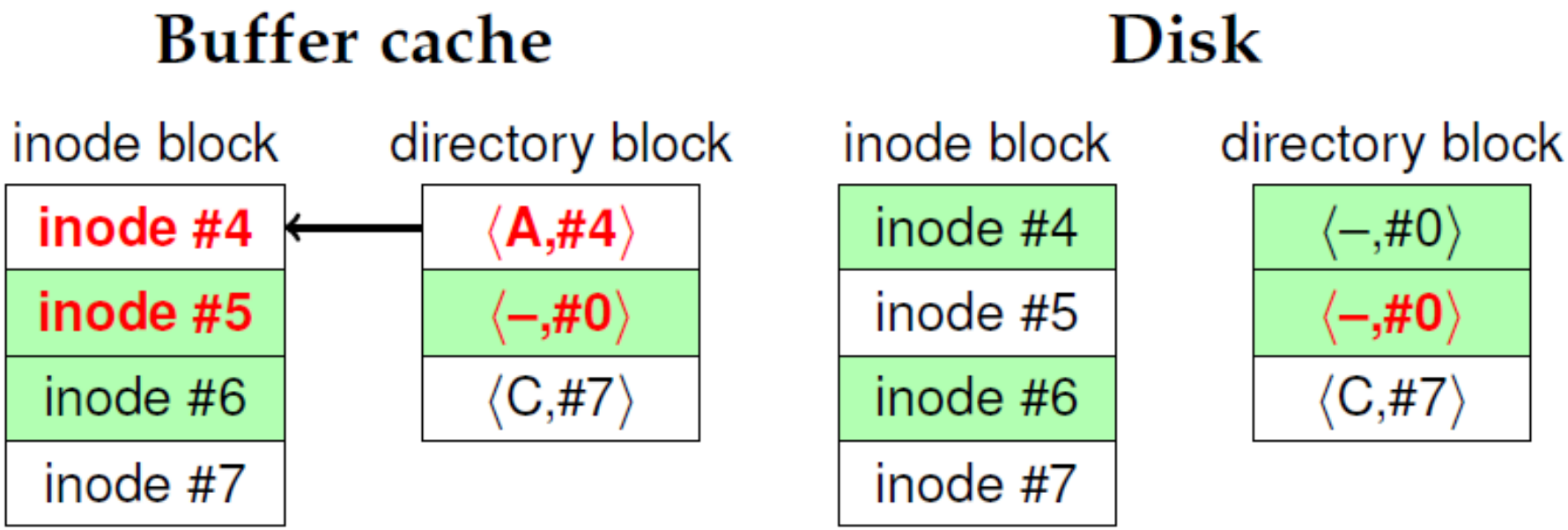
- Write blocks in any order
- Keep track of dependencies
- When writing a block, unroll any changes you can't yet commit to disk

Breaking Dependencies



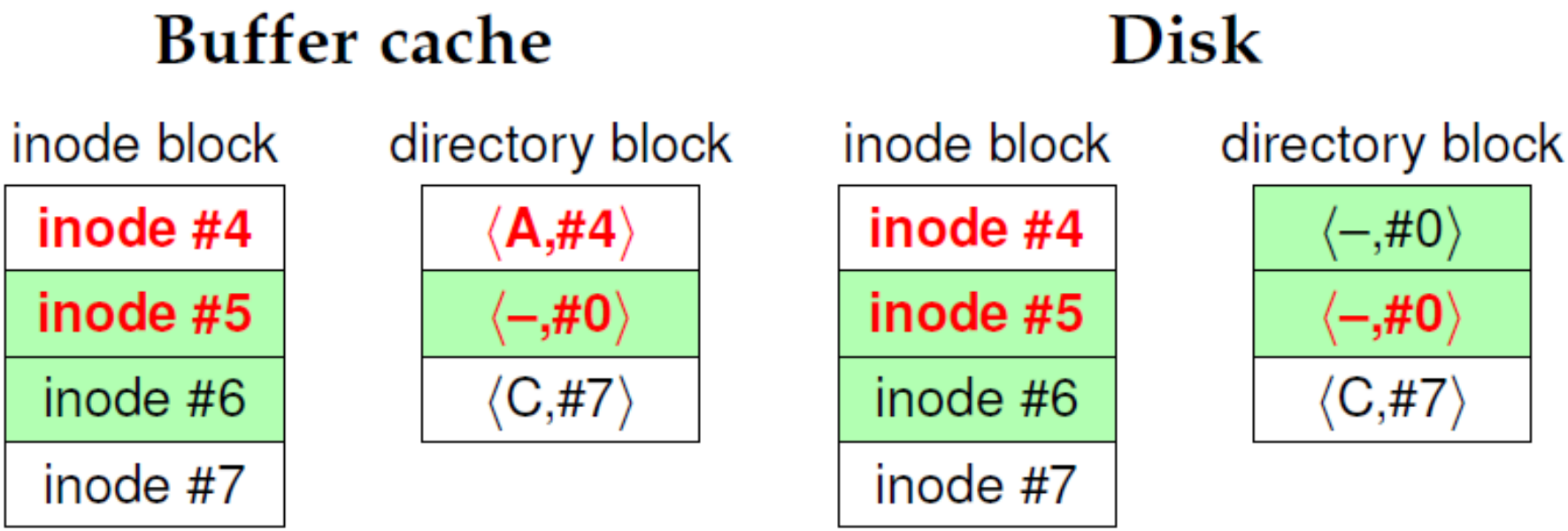
- Created file A and deleted file B
- Now say we decide to write directory block
- Can’t write file name A to disk – has dependee

Breaking Dependencies



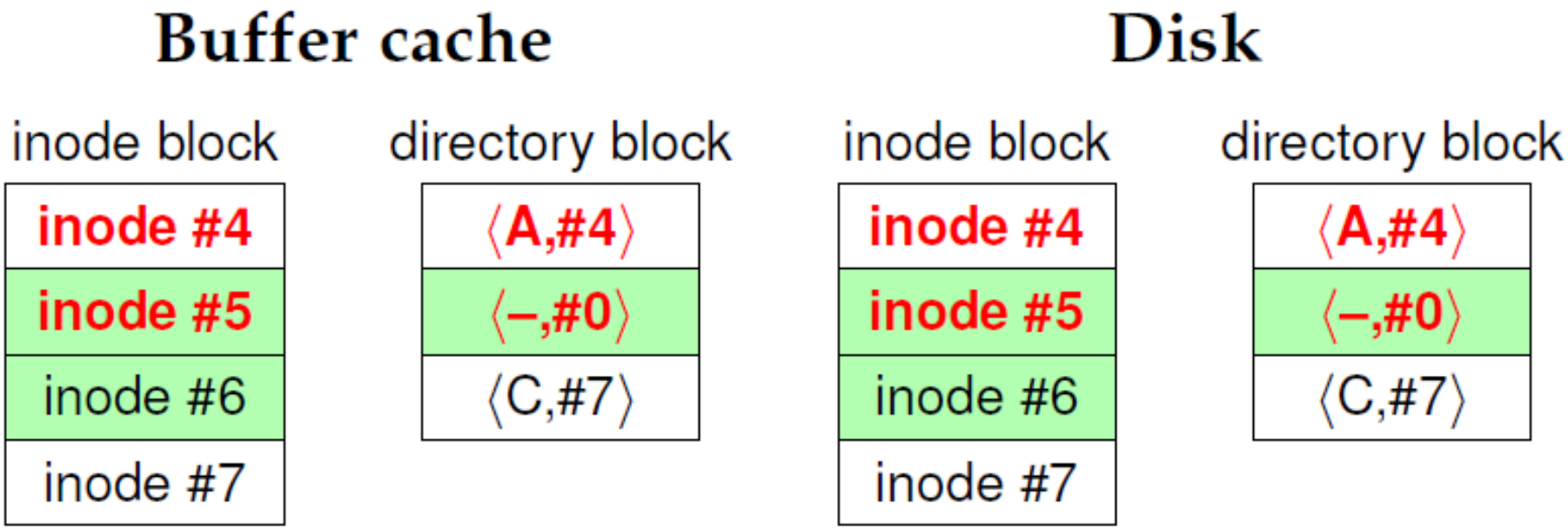
- Undo file A before writing dir block to disk
- But now inode block has no dependees
  - Can safely write inode block to disk as-is

Breaking Dependencies



- Now inode block clean (same in memory/disk)
- But have to write directory block a second time

Breaking Dependencies



- All data stably on disk
- Crash at any point would have been safe

Soft Update Issues

- fsync: May flush directory entries, etc.
- unmount: Some disk buffers flushed multiple times
- Deleting directory tree fast!
  - unlink doesn’t need to read file inodes synchronously
  - Careful with memory!
- Useless write backs
  - Syncer flushes dirty buffers to disk every 30 seconds
  - Writing all at once means many dependency issues
  - Fix syncer to write blocks one at a time
  - LRU buffer eviction needs to know about dependencies

Soft Updates: fsck

- Split into foreground + background parts
- Foreground must be done before remounting
  - Ensure per-cylinder summary makes sense
  - Recompute free block/inode counts from bitmap (fast)
  - Leave FS consistent but may leak disk space
- Background does traditional fsck
  - Done on a snapshot while system is running
  - A syscall allows fsck to have FFS patch file system
  - Snapshot deleted once complete
- Much shorter downtimes than traditional fsck

Outline

- Overview
- FFS Review and Details
- Crash Recoverability
- Soft Updates
- **Journaling**
- LFS/WAFL

Journaling

- Idea: Use Write-Ahead Log to Journal Metadata
  - Reserve portion of the disk
  - Write operation to log, then to the disk
  - After a crash/reboot, replay the log (efficient)
  - May redo already committed changes, but doesn’t miss
- Physical Journal: Both data + metadata written
  - Induces extra IO overhead to disk
  - Simple to implement and safe for application
- Logical Journal: Only metadata written to journal
  - Metadata journaling
  - More complex implementation

Journaling (Continued)

- Performance Advantage:
  - Log is a linear region of the disk
  - Multiple operations can be logged
  - Final operations written asynchronously
- Journals typically large (~1 GB)
  - Copy of disk blocks written
  - Requires high throughput
- Example: Deleting directory tree
  - Journal all freed blocks, changed directory blocks, etc
  - Return to user
  - In background we can write out changes in any order

SGI XFS: [Sweeney]

- Main idea:
  - Big disks, files, and large # of files, 64-bit everything
  - Maintain very good performance
- Break disk up into Allocation Groups (AGs)
  - 0.5 – 4 GB regions of a disk
  - Similar to Cylinder-Groups but for different purpose: AGs too large to minimize seek times
  - AGs have no fixed # of inodes
- Advantages:
  - Parallelize allocation, and data structures for multicore
  - Used on super computers with many cores
  - 32-bit pointers within AGs for size

XFS B+ Trees

- XFS makes extensive use of B+ Trees
  - Indexed data structure stores ordered keys & values
  - Keys have defined ordering
- Three main operations O(log(n))
  - Insert a new <key, value> pair
  - Delete <key, value> pair
  - Retrieve closest <key, value> to target key k
  - See any algorithms book for details
- Used to:
  - Free space management
  - Extended attributes
  - Extent map for files: file offset to <start, length>

B+ Trees Continued

- Space Allocation:
  - Two B+ Trees: Sorted by length, Sorted by address
  - Easily find nearby blocks easily (locality)
  - Easily find large extents for large files (best fit)
  - Easily coalesce adjacent free regions
- Journaling enables complex atomic operations
  - First write metadata changes to log on-disk
  - Apply changes to disk
  - On Crash:
    - If log is incomplete, log will be discarded
    - Otherwise replay log

Journaling vs. Soft Updates

- Limitations of Soft Updates
  - High complexity and very tied to FFS data format
  - Metadata updates may proceed out of order
    - create A, create B, crash – maybe only B exists after reboot
  - Still need slow background fsck
- Limitations of Journaling
  - Disk write required for every metadata operation
    - Create-then-delete may not require I/Os with soft updates
  - Possible contention for the end of log on multiprocessor
  - *fsync* must sync other operations' metadata to log
- Can we get the best of both worlds?

Soft Updates + Journaling [McKusick]

- Example of minimalist Metadata Journaling
- Journal resource allocations and future references
- fsck can recover file system state from journal

- 16 Mb journal (instead of GBs for data journals)
- 32 byte journal entries
- 27 types of journal entries

- Fast fsck takes ~2 seconds
- Less bandwidth and cost than normal journaling

Outline

- Overview
- FFS Review and Details
- Crash Recoverability
- Soft Updates
- Journaling
- **LFS/WAFL**

Log File System [Mendel]

- Main Idea: Only have a journal!
- Fast writing
  - Fragmentation from long lived data may be a problem
- Slow reading (lots of seeks)
  - Finding the head of the file system may take time
- Log cleaning:
  - A process needs to periodically free old blocks from the file system
- SSDs reduce seek overhead – LFS practical again

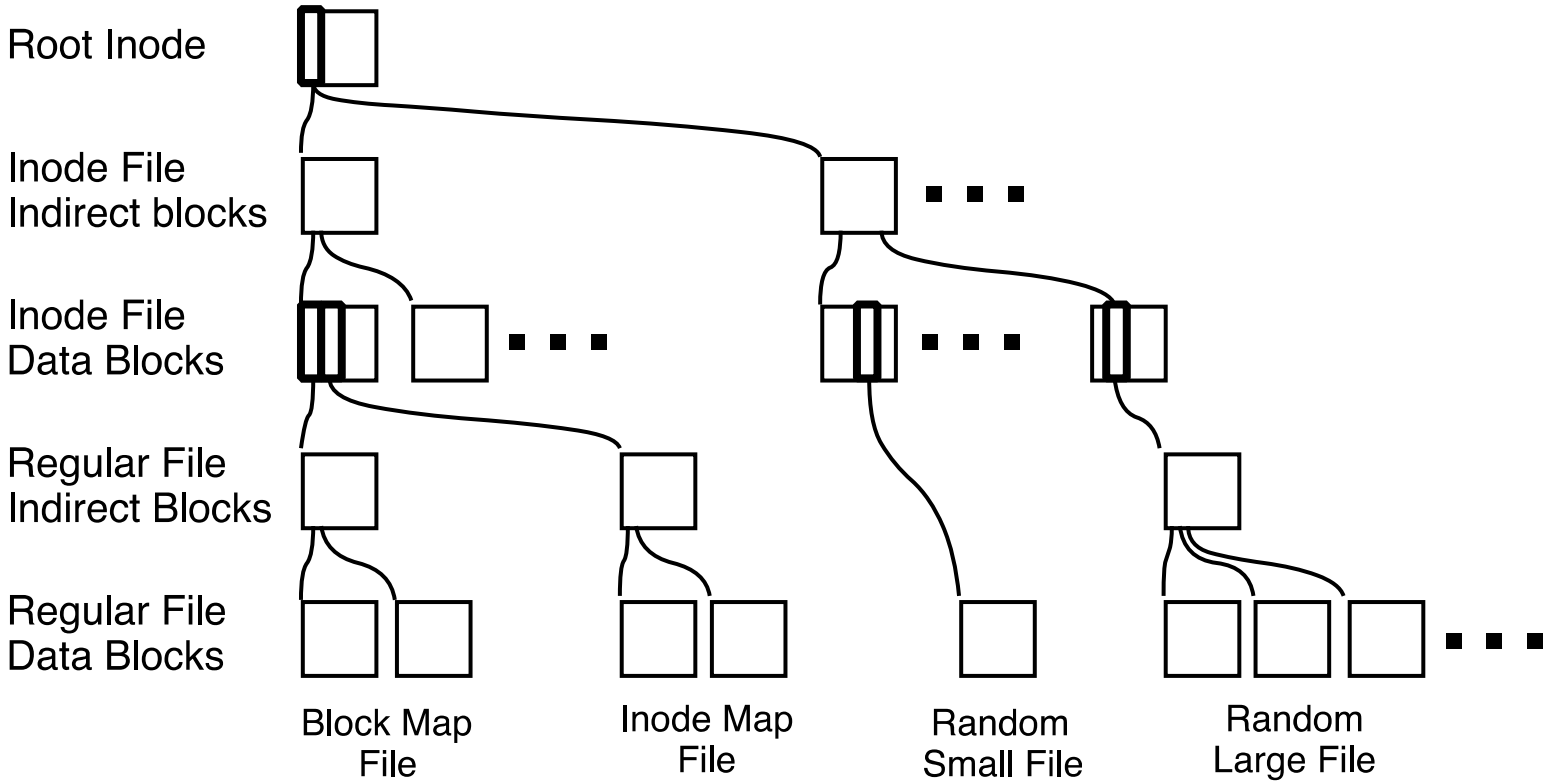
Write Anywhere File Layout (WAFL)

- Copy-on-Write File System
- Inspired ZFS, HAMMER, btrfs
- Core Idea: Write whole snapshots to disk
- Snapshots are virtually free!
- Snapshots accessible from .snap directory in root

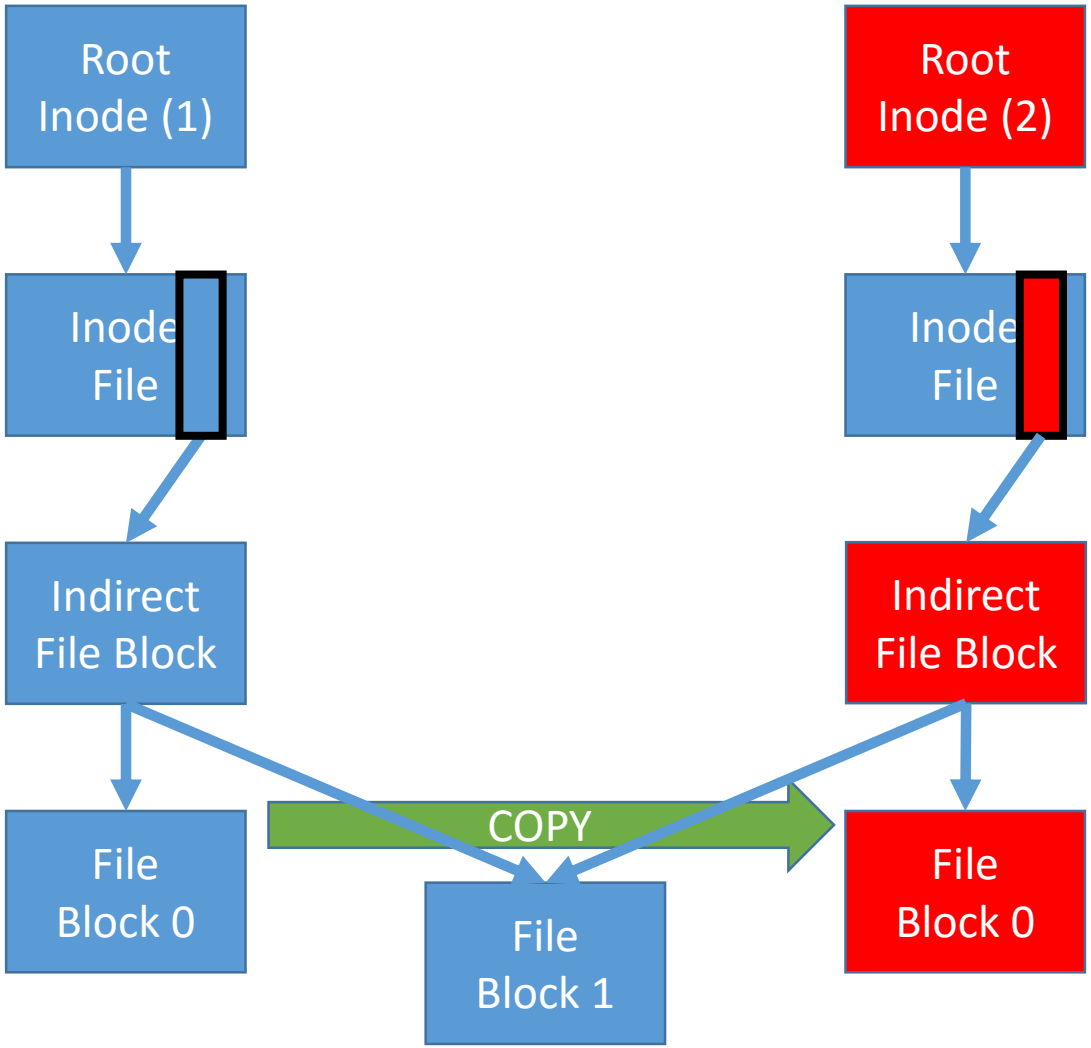
```
spike% ls -lut .snapshot/*/todo
-rw-r--r-- 1 hitz  52880 Oct 15 00:00
.snapshot/nightly.0/todo
-rw-r--r-- 1 hitz  52880 Oct 14 19:00
.snapshot/hourly.0/todo
-rw-r--r-- 1 hitz  52829 Oct 14 15:00
.snapshot/hourly.1/todo
```

WAFL: Detailed View

- Only root at fixed location
- FS structures are accessed through Inodes
  - Block allocation, Inode Table, Files, Directories



WAFL: Example

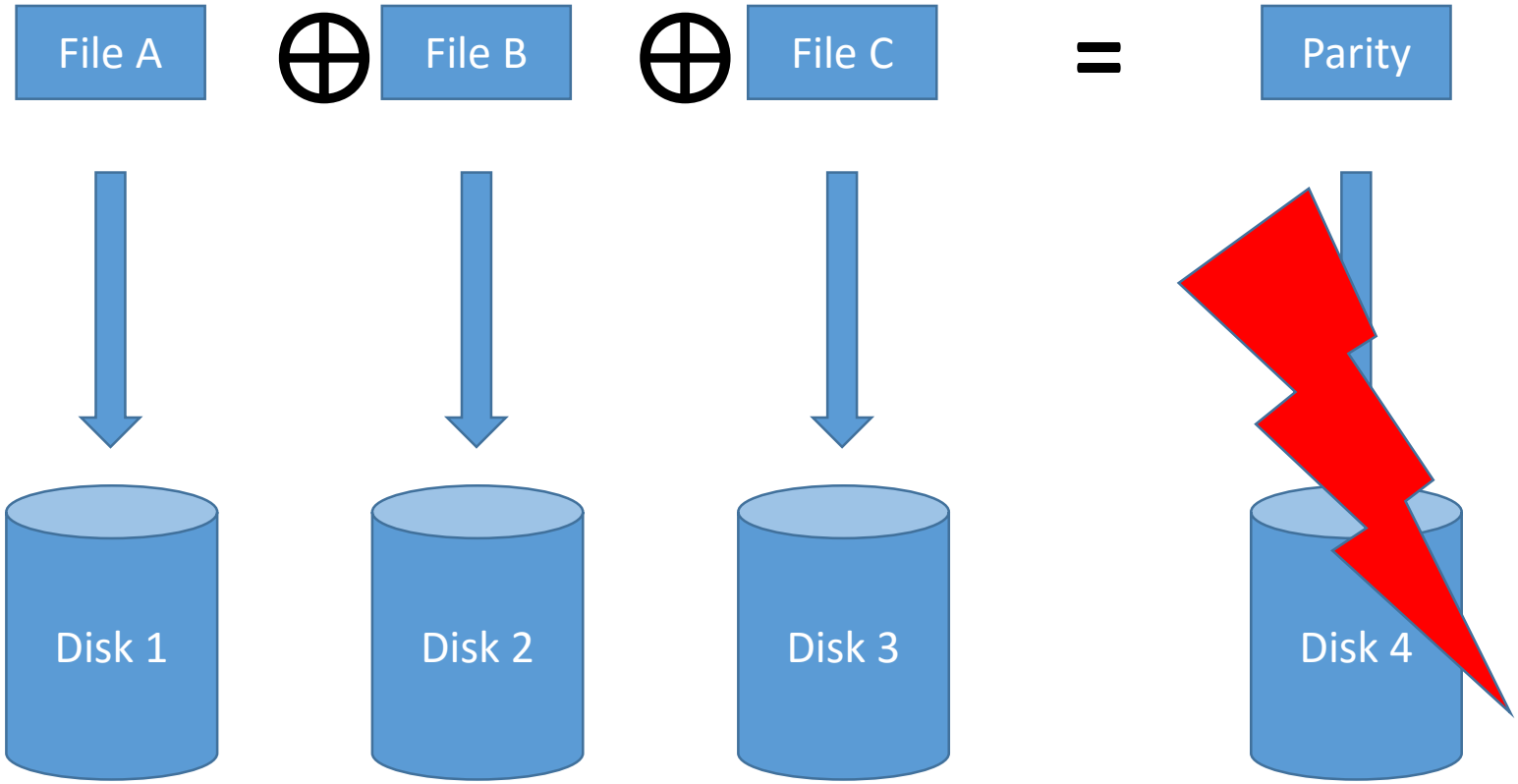


- Consistency:
  - On crash recovery find a snapshot that has been fully committed to disk
  - Reclaim space after whole snapshot written to disk
- Persistent Snapshots:
  - Save root inode and do not reclaim data older than latest snapshot

ZFS

- Copy-on-Write functions similar to WAFL
- Integrates Volume Manager & File System
  - Software RAID without the write hole
- Integrates File System & Buffer Management
  - Advanced prefetching: strided patterns etc.
  - Use Adaptive Replacement Cache (ARC) instead of LRU
- File System reliability
  - Check summing of all data and metadata
  - Redudant Metadata

RAID Write Hole



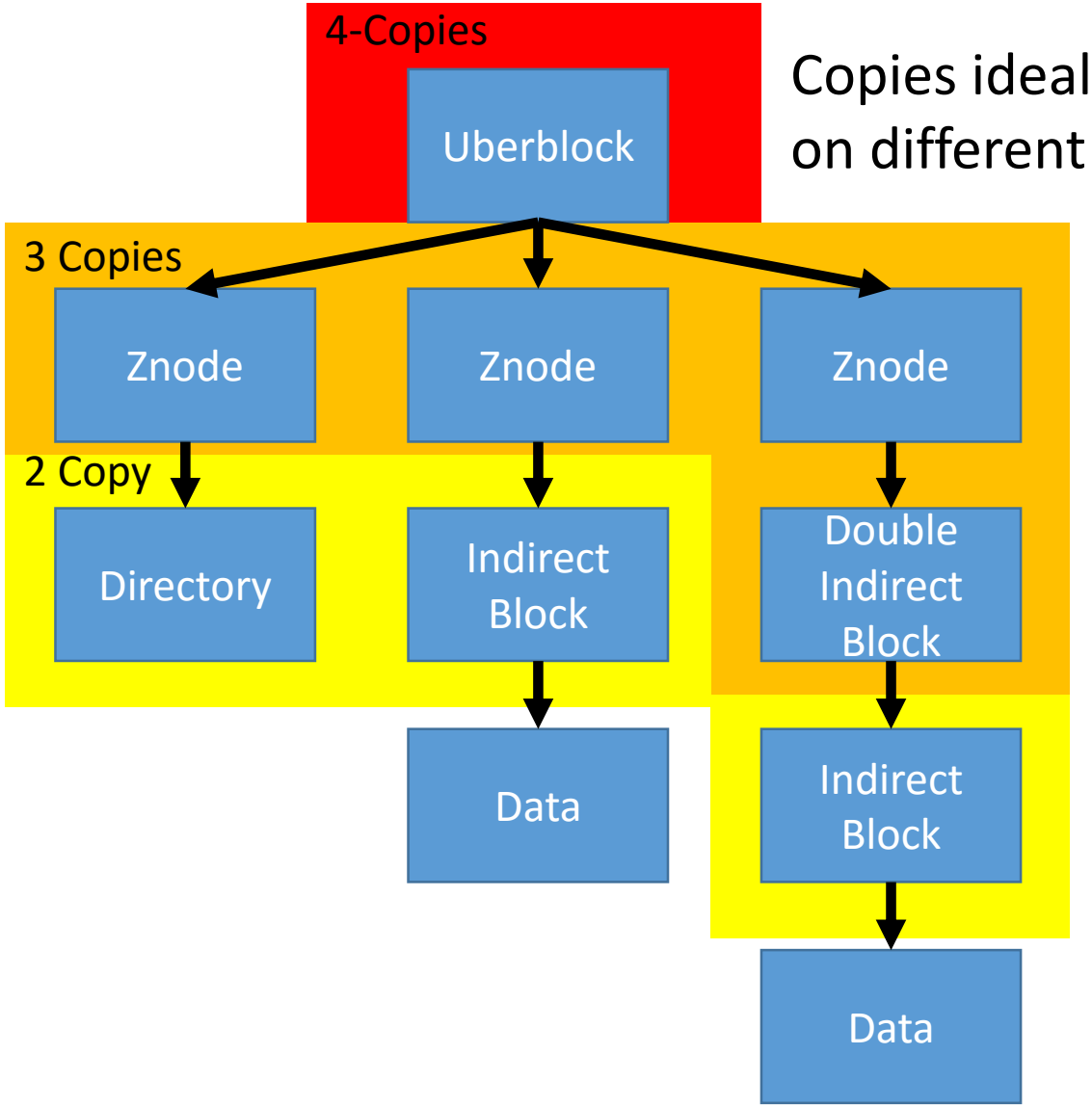
Result: No way to distinguish between this and silent data corruption

ZFS Volume Management

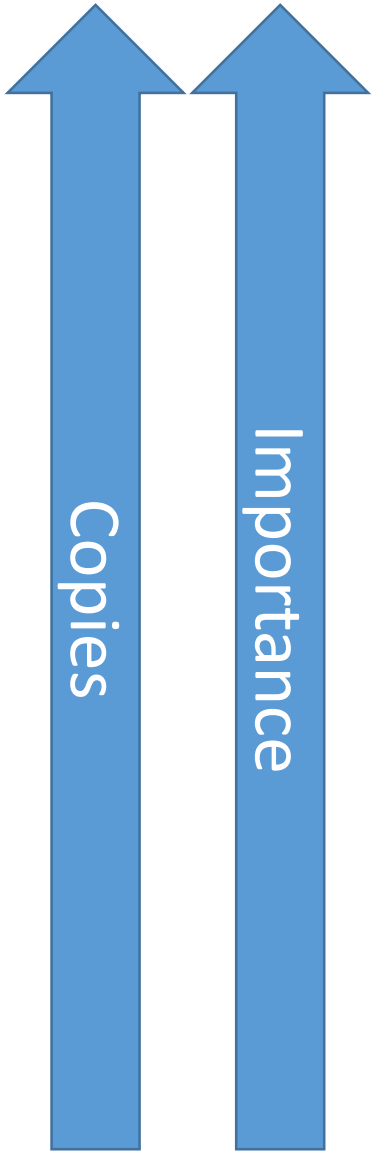
- Volume management and RAID part of ZFS
- Checksums can identify:
  - Silent data corruption
  - RAID striped data not written in entirety

- File system verifies recent snapshots on mount to fix write hole issues
- If only one drive’s data is corrupt we don’t lose data

Redudant Metadata [zfs(8)]



Copies ideally placed on different disks



Summary

- Performance & Recoverability Trade-off
- fsck: Metadata overhead, fsck very slow
- Soft Updates: Background recovery, low overhead
- Journaling: Fast recovery, low overhead
- Soft Updates + Journaling
  - Fast recovery, low overhead
- LFS/WAFL
  - Fast writing/Higher read costs
  - Almost no recovery time!

Cluster/Distributed File System

- Shared Disk Cluster File Systems
  - VxFS (JFS in HP-UX) – Clusters
    - Network based locking
  - VMFS – VMware's clustering FS
    - Uses SCSI reservations to coordinate between nodes
- Local Disk Cluster
  - GFS, Ceph, Lustre, Windows DFS
- Both
  - IBM GPFS