CS162
Operating Systems and Systems Programming
Lecture 20

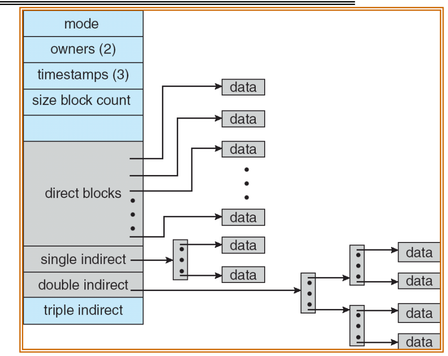
Filesystems (Con't)
Reliability, Transactions

April 14th, 2020

Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: Multilevel Indexed Files (Original 4.1 BSD)

- Sample file in multilevel indexed format:
  - 10 direct ptrs, 1K blocks
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data

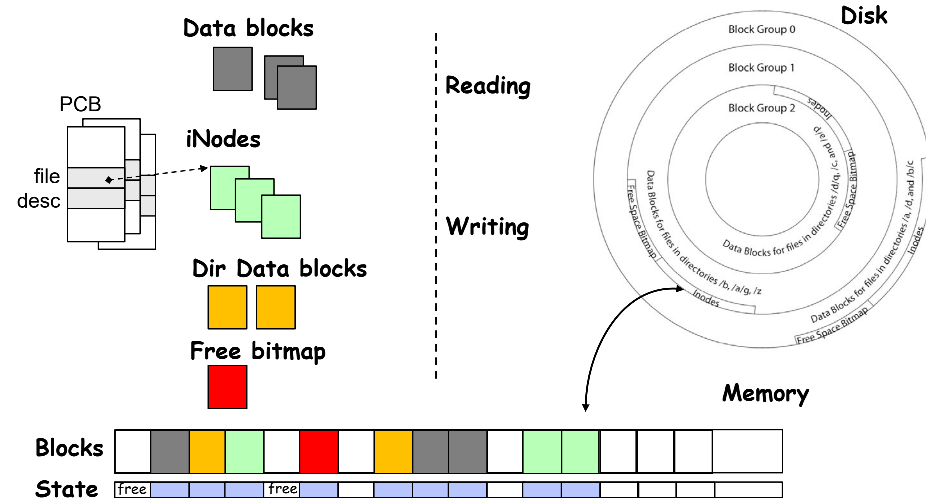


- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks (lead to 4.2 Fast File System Optimizations)
- Ext2/3 (Linux):
  - 12 direct ptrs, triply-indirect blocks, settable block size (4K is common)

Recall: Buffer Cache

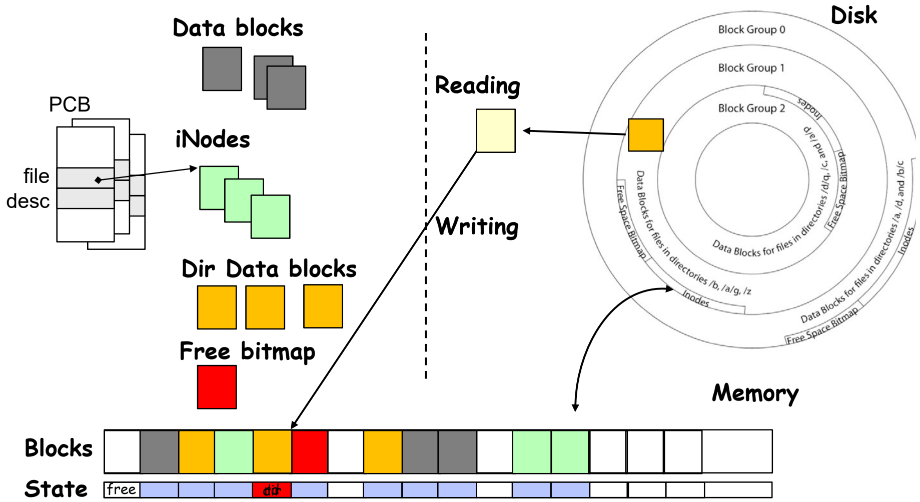
- Kernel must copy disk blocks to main memory to access their contents and write them back if modified
  - Could be data blocks, inodes, directory contents, etc.
  - Possibly dirty (modified and not written back)
- Key Idea: Exploit locality by caching disk data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks yet on disk)

File System Buffer Cache



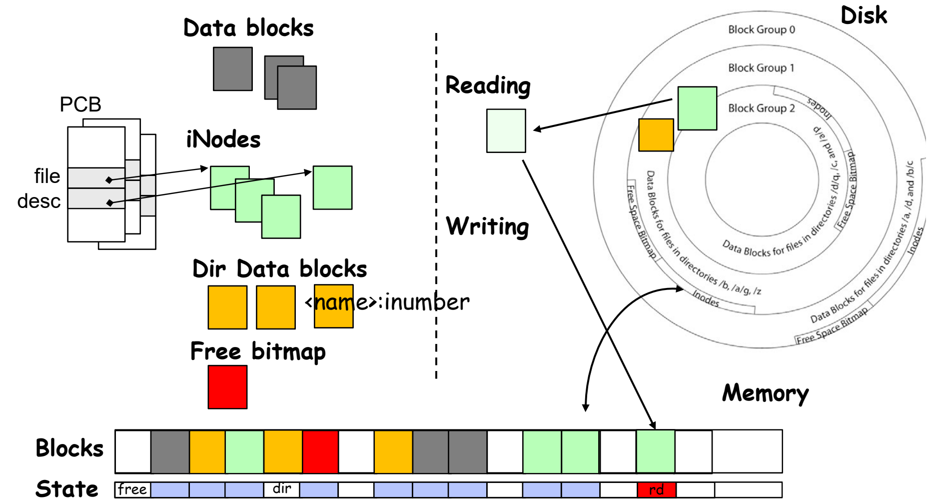
- OS implements a cache of disk blocks for efficient access to data, directories, inodes, freemap

File System Buffer Cache: open



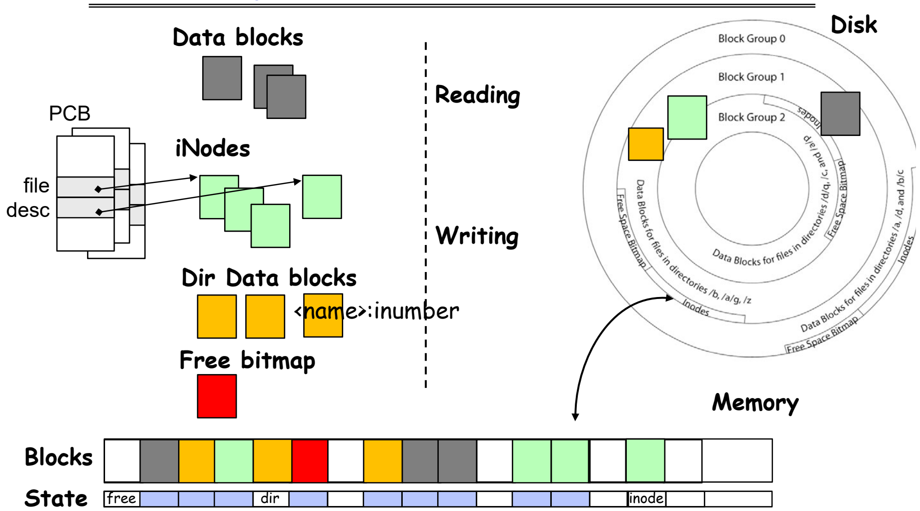
- {load block of directory; search for map}**+** ;

File System Buffer Cache: open



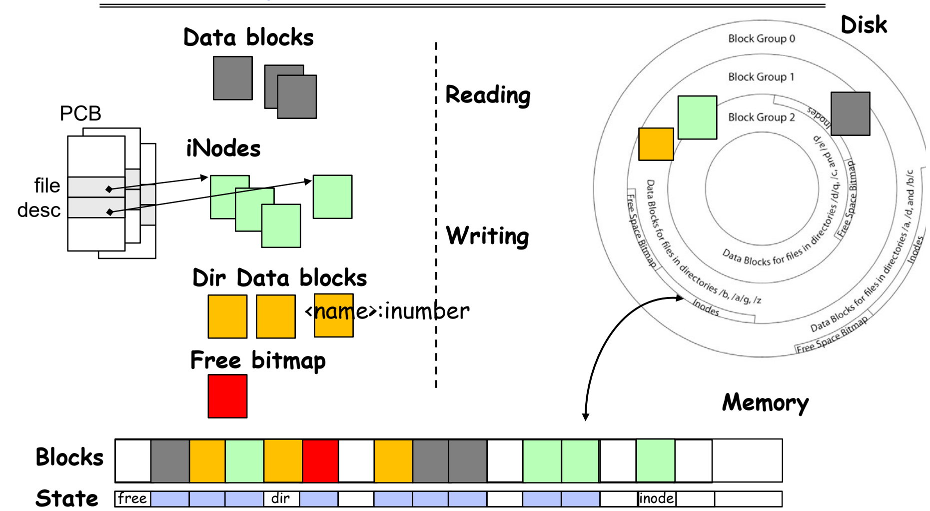
- {load block of directory; search for map}+ ; Load inode ;
- Create reference via open file descriptor

File System Buffer Cache: Read?



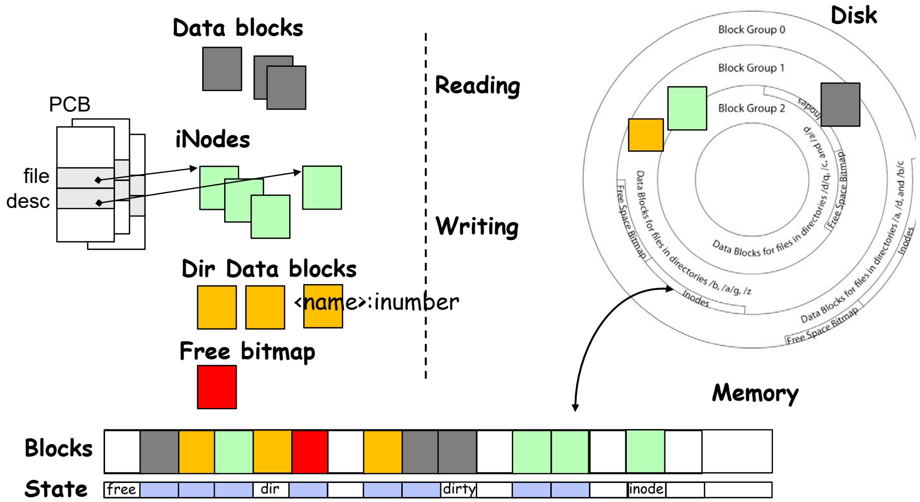
- From inode, traverse index structure to find data block; load data block; copy all or part to read data buffer

File System Buffer Cache: Write?



- Process similar to read, but may allocate new blocks (update free map), blocks need to be written back to disk; inode?

File System Buffer Cache: Eviction?



- Blocks being written back to disc go through a transient state

Buffer Cache Discussion

- Implemented entirely in OS software
  - Unlike memory caches and TLB
- Blocks go through transitional states between free and in-use
  - Being read from disk, being written to disk
  - Other processes can run, etc.
- Blocks are used for a variety of purposes
  - inodes, data for dirs and files, freemap
  - OS maintains pointers into them
- Termination – e.g., process exit – open, read, write
- Replacement – what to do when it fills up?

File System Caching

- Replacement policy? LRU
  - Can afford overhead full LRU implementation
  - Advantages:
    - » Works very well for name translation
    - » Works well in general as long as memory is big enough to accommodate a host's working set of files.
  - Disadvantages:
    - » Fails when some application scans through file system, thereby flushing the cache with data used only once
    - » Example: `find . -exec grep foo {} \;`
- Other Replacement Policies?
  - Some systems allow applications to request other policies
  - Example, 'Use Once':
    - » File system can discard blocks as soon as they are used

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
  - Too much memory to the file system cache ⇒ won't be able to run many applications at once
  - Too little memory to file system cache ⇒ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
- Read Ahead Prefetching: fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
  - Elevator algorithm can efficiently interleave groups of prefetches from concurrent applications
  - How much to prefetch?
    - » Too many imposes delays on requests by other applications
    - » Too few causes many seeks (and rotational delays) among concurrent file requests

Delayed Writes

- Delayed Writes: Writes to files not immediately sent to disk
  - So, Buffer Cache is a write-back cache
- `write()` copies data from user space buffer to kernel buffer
  - Enabled by presence of buffer cache: can leave written file blocks in cache for a while
  - Other apps **read data from cache** instead of disk
  - Cache is *transparent* to user programs
- Flushed to disk periodically
  - In Linux: kernel threads flush buffer cache very 30 sec. in default setup
- Disk scheduler can efficiently order lots of requests
  - Elevator Algorithm can rearrange writes to avoid random seeks

Delayed Writes

- Delay block allocation: May be able to allocate multiple blocks at same time for file, keep them contiguous
- Some files never actually make it all the way to disk
  - Many short-lived files
- But what if system crashes before buffer cache block is flushed to disk?
- And what if this was for a directory file?
  - Lose pointer to inode
- **file systems need recovery mechanisms**

Important "ilities"

- Availability: the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

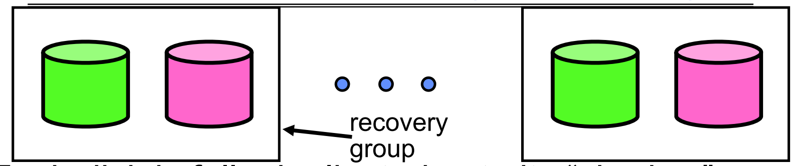
How to Make File System Durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - Use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache
- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: independence of failure
    - » Could put copies on one disk, but if disk head fails…
    - » Could put copies on different disks, but if server fails…
    - » Could put copies on different servers, but if building is struck by lightning….
    - » Could put copies on servers in different continents…

RAID: Redundant Arrays of Inexpensive Disks

- Classified by David Patterson, Garth A. Gibson, and Randy Katz here at UCB in 1987
  - Classic paper was first to evaluate multiple schemes
- Data stored on multiple disks (redundancy)
  - Berkeley researchers were looking for alternatives to big expensive disks
  - Redundancy necessary because cheap disks were more error prone
- Either in software or hardware
  - In hardware case, done by disk controller; file system may not even know that there is more than one disk in use
- Initially, five levels of RAID (more now)

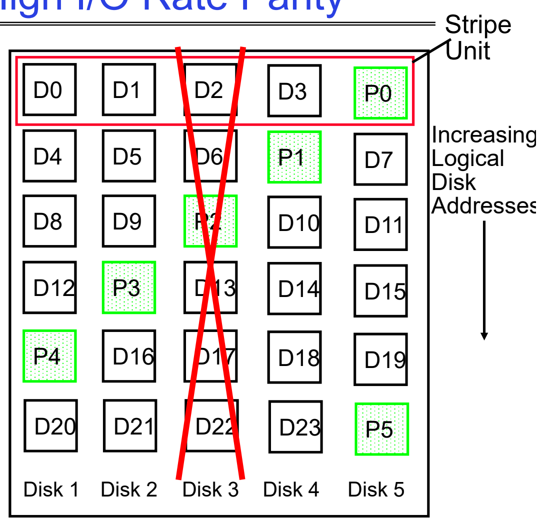
RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its “shadow”
  - For high I/O rate, high availability environments
  - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
  - Logical write = two physical writes
  - Highest bandwidth when disk heads and rotation fully synchronized (hard to do exactly)
- Reads may be optimized
  - Can have two independent reads to same data
- Recovery:
  - Disk failure ⇒ replace disk and copy data to new disk
  - Hot Spare: idle disk already attached to system to be used for immediate replacement

RAID 5+: High I/O Rate Parity

- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data bocks in stripe
  - $P0=D0\oplus D1\oplus D2\oplus D3$
  - Can destroy any one disk and still reconstruct data
  - Suppose Disk 3 fails, then can reconstruct: $D2=D0\oplus D1\oplus D3\oplus P0$



- Can spread information widely across internet for durability
  - RAID algorithms work over geographic scale

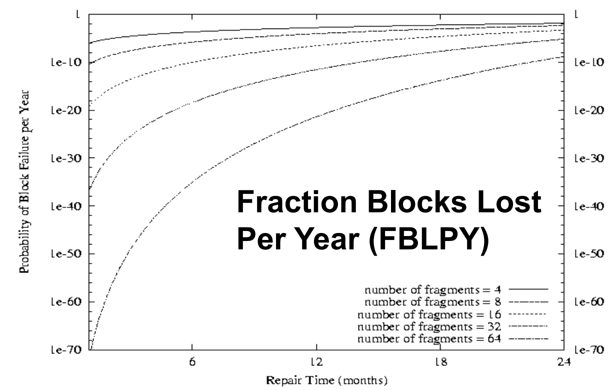
Allow more disks to fail!

- In general: RAIDX is an “erasure code”
  - Must have ability to know which disks are bad
  - Treat missing disk as an “Erasure”
- Today, Disks so big that: RAID 5 not sufficient!
  - Time to repair disk sooooo long, another disk might fail in process!
  - “RAID 6” – allow 2 disks in replication stripe to fail
- But – must do something more complex that just XORing together blocks!
  - Already used up the simple XOR operation across disks
- Simple option: Check out EVENODD code in readings
  - Will generate one additional check disks to support RAID 6
- More general option for general erasure code: Reed-Solomon codes
  - Based on polynomials in $GF(2^k)$ (I.e. k-bit symbols)
    - » Gailois Field is finite version of real numbers
  - Data as coefficients (a_j), code space as values of polynomial:
    - » $P(x)=a_0+a_1x^1+\ldots a_{m-1}x^{m-1}$
    - » Coded: P(0),P(1),P(2)….,P(n-1)
  - Can recover polynomial (i.e. data) as long as get any m of n; allows n-m failures!

Allow more disks to fail! (Con't)

- How to use Reed-Solomon code in practice?
  - Each coefficient has a fixed (k) number of bits. So, must encode with symbols that size
  - Example: k=16 bit symbols, m=4, encoding 16x4 bits at a time
    - » Take original data, split into 4 chunks. On each encoding step, grab 16 bits from each chunk to use as coefficients
    - » Each data point yields a 16-bit symbol, which you distributed to final encoded chunks
  - (better version of Reed-Solomon code for erasure channels is the "Cauchy Reed-Solomon" code; it is isomorphic to the version here)
- Examples (with k=16):
  - Suppose have 6 disks, want to tolerate 2 failures
    - » Split data into 4 chunks, encode 16 bits from each chunk at a time, by generating 6 points (of 16 bits) on 3rd-degree polynomial
    - » Distribute data from polynomial to 6 disks – each disk will ultimately hold data that is ¼ size of original data
    - » Can handle 2 lost disks for 50% overhead
  - More interesting extreme for Internet-level replication:
    - » Split data into 4 chunks, produce 16 chunks
    - » Each chunk is ¼ total size of original data, Overhead = factor of 4
    - » But – only need 4 of 16 fragments! REALLY DURABLE!

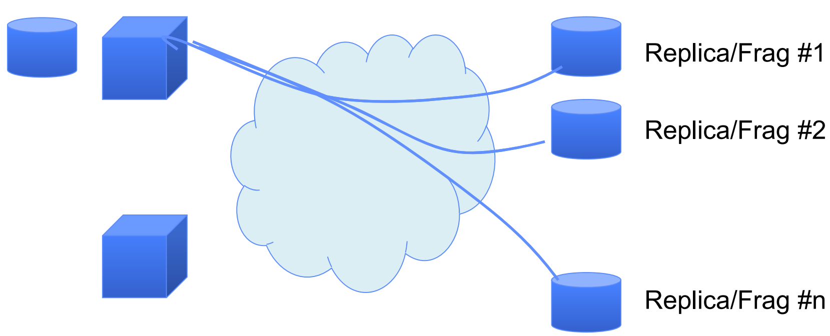
Use of Erasure Coding in general: High Durability/overhead ratio!



- Exploit law of large numbers for durability!
- 6 month repair, FBLPY with 4x increase in total size of data:
  - Replication (4 copies): 0.03
  - Fragmentation (16 of 64 fragments needed): 10^{-35}

Higher Durability/Reliability through Geographic Replication

- Highly durable – hard to destroy all copies
- Highly available for reads
  - Simple replication: read any copy
  - Erasure coded: read m of n
- Low availability for writes
  - Can't write if any one replica is not up
  - Or – need relaxed consistency model
- Reliability? – availability, security, durability, fault-tolerance



File System Reliability: (Difference from Block-level reliability)

- What can happen if disk loses power or software crashes?
  - Some operations in progress may complete
  - Some operations in progress may be lost
  - Overwrite of a block may only partially complete
- Having RAID doesn't necessarily protect against all such failures
  - No protection against writing bad state
  - What if one disk of RAID group not written?
- File system needs durability (as a minimum!)
  - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
  - inode, indirect block, data block, bitmap, …
  - With sector remapping, single update to physical disk block can require multiple (even lower level) updates to sectors
- At a physical level, operations complete one at a time
  - Want concurrent operations for performance
- How do we guarantee consistency regardless of when crash occurs?

Threats to Reliability

- Interrupted Operation
  - Crash or power failure in the middle of a series of related updates may leave stored data in an inconsistent state
  - Example: transfer funds from one bank account to another
  - What if transfer is interrupted after withdrawal and before deposit?
- Loss of stored data
  - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

Reliability Approach #1: Careful Ordering

- Sequence operations in a specific order
  - Careful design to allow sequence to be interrupted safely
- Post-crash recovery
  - Read data structures to see if there were any operations in progress
  - Clean up/finish as needed
- Approach taken by
  - FAT and FFS (fsck) to protect filesystem structure/metadata
  - Many app-level recovery schemes (e.g., Word, emacs autosaves)

FFS: Create a File

Normal operation:

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks and inodes
- Update directory with file name → inode number
- Update modify time for directory

Recovery:

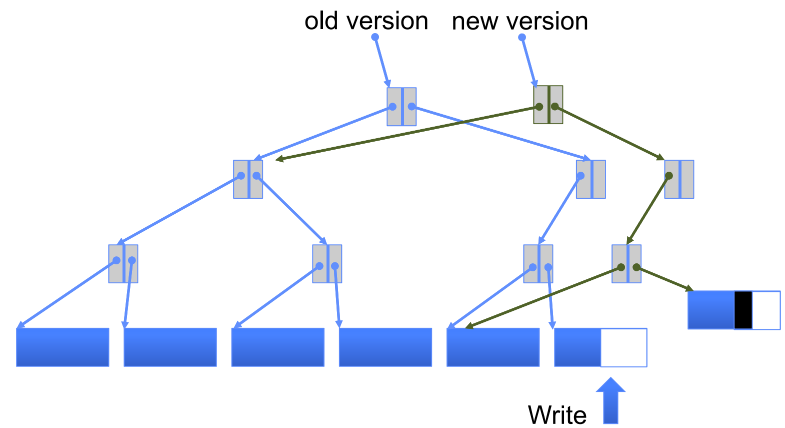
- Scan inode table
- If any unlinked files (not in any directory), delete or put in lost & found dir
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times

Time proportional to disk size

Reliability Approach #2: Copy on Write File Layout

- To update file system, write a new version of the file system containing the update
  - Never update in place
  - Reuse existing unchanged disk blocks
- Seems expensive! But
  - Updates can be batched
  - Almost all disk writes can occur in parallel
- Approach taken in network file server appliances
  - NetApp's Write Anywhere File Layout (WAFL)
  - ZFS (Sun/Oracle) and OpenZFS

COW with Smaller-Radix Blocks



- If file represented as a tree of blocks, just need to update the leading fringe

ZFS and OpenZFS

- Variable sized blocks: 512 B – 128 KB
- Symmetric tree
  - Know if it is large or small when we make the copy
- Store version number with pointers
  - Can create new version by adding blocks and new pointers
- Buffers a collection of writes before creating a new version with them
- Free space represented as tree of extents in each block group
  - Delay updates to freespace (in log) and do them all when block group is activated

More General Reliability Solutions

- Use Transactions for atomic updates
  - Ensure that multiple related updates are performed atomically
  - i.e., if a crash occurs in the middle, the state of the systems reflects either all or none of the updates
  - Most modern file systems use transactions internally to update filesystem structures and metadata
  - Many applications implement their own transactions
- Provide Redundancy for media failures
  - Redundant representation on media (Error Correcting Codes)
  - Replication across media (e.g., RAID disk array)

Transactions

- Closely related to critical sections for manipulating shared data structures
- They extend concept of atomic update from memory to stable storage
  - Atomically update multiple persistent data structures
- Many ad-hoc approaches
  - FFS carefully ordered the sequence of updates so that if a crash occurred while manipulating directory or inodes the disk scan on reboot would detect and recover the error (fsck)
  - Applications use temporary files and rename

Key Concept: Transaction

- An atomic sequence of actions (reads/writes) on a storage system (or database)
- That takes it from one consistent state to another

consistent state 1 $\xrightarrow{\text{transaction}}$ consistent state 2

Typical Structure

- Begin a transaction – get transaction id
- Do a bunch of updates
  - If any fail along the way, roll-back
  - Or, if any conflicts with other transactions, roll-back
- Commit the transaction

"Classic" Example: Transaction

```
BEGIN;    --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance - 100.00 WHERE
   name = 'Alice';

 UPDATE branches SET balance = balance - 100.00 WHERE
   name = (SELECT branch_name FROM accounts WHERE name
   = 'Alice');

 UPDATE accounts SET balance = balance + 100.00 WHERE
   name = 'Bob';

 UPDATE branches SET balance = balance + 100.00 WHERE
   name = (SELECT branch_name FROM accounts WHERE name
   = 'Bob');
COMMIT;    --COMMIT WORK
```

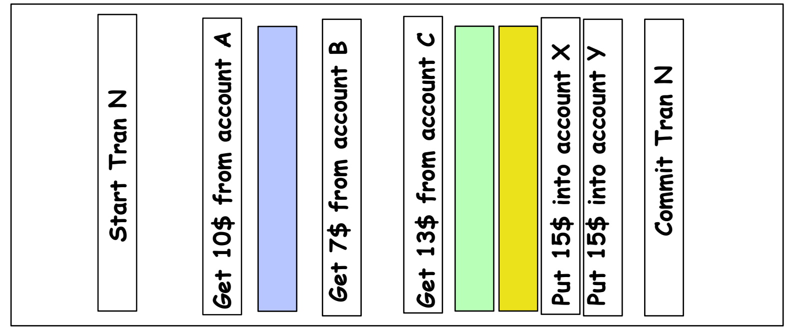
Transfer $100 from Alice's account to Bob's account

The ACID properties of Transactions

- Atomicity: all actions in the transaction happen, or none happen
- Consistency: transactions maintain data integrity, e.g.,
  - Balance cannot be negative
  - Cannot reschedule meeting on February 30
- Isolation: execution of one transaction is isolated from that of all others; no problems from concurrency
- Durability: if a transaction commits, its effects persist despite crashes

Concept of a log

- One simple action is atomic – write/append a basic item
- Use that to seal the commitment to a whole series of actions



Transactional File Systems

- Better reliability through use of log
  - All changes are treated as *transactions*
  - A transaction is *committed* once it is written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between "Log Structured" and "Journaled"
  - In a Log Structured filesystem, data stays in log form
  - In a Journaled filesystem, Log used for recovery
- Journaling File System
  - Applies updates to system metadata using transactions (using logs, etc.)
  - Updates to non-directory files (i.e., user stuff) can be done in place (without logs), full logging optional
  - Ex: NTFS, Apple HFS+, Linux XFS, JFS, ext3, ext4
- Full Logging File System
  - All updates to disk are done in transactions

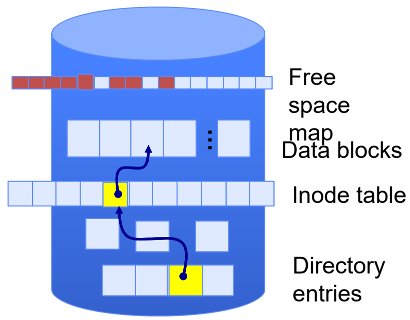
Journaling File Systems

- Instead of modifying data structures on disk directly, write changes to a journal/log
  - Intention list: set of changes we intend to make
  - Log/Journal is append-only
  - Single commit record commits transaction
- Once changes are in the log, it is safe to apply changes to data structures on disk
  - Recovery can read log to see what changes were intended
  - Can take our time making the changes
    - » As long as new requests consult the log first
- Once changes are copied, safe to remove log
- But, …
  - If the last atomic action is not done … poof … all gone
- Basic assumption:
  - Updates to sectors are atomic and ordered
  - Not necessarily true unless very careful, but key assumption

Example: Creating a File

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

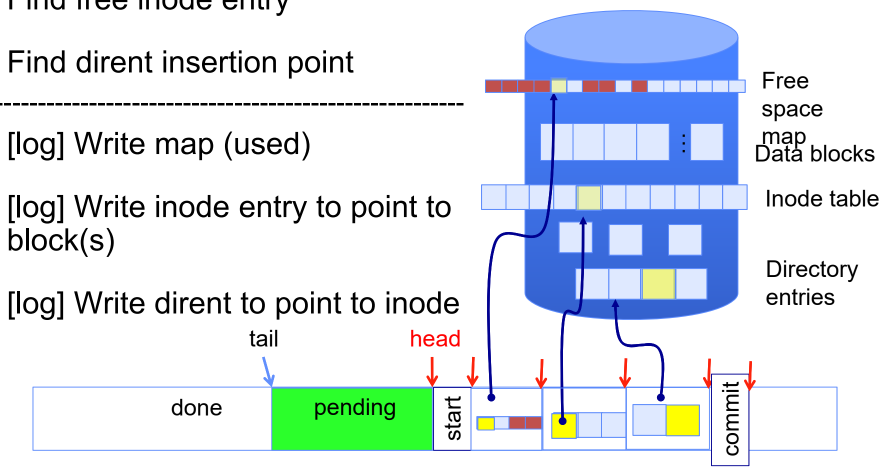
- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



Ex: Creating a file (as a transaction)

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

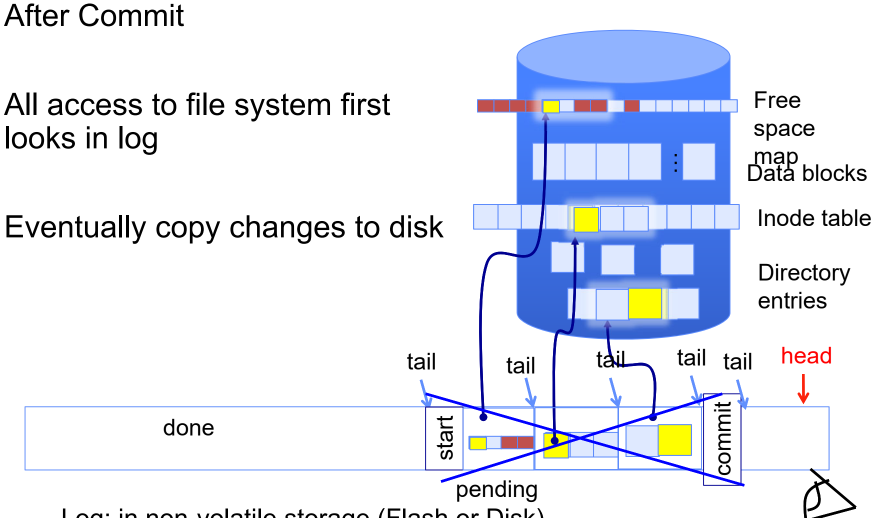
- [log] Write map (used)
- [log] Write inode entry to point to block(s)
- [log] Write dirent to point to inode



Log: in non-volatile storage (Flash or on Disk)

“Redo Log “ – Replay Transactions

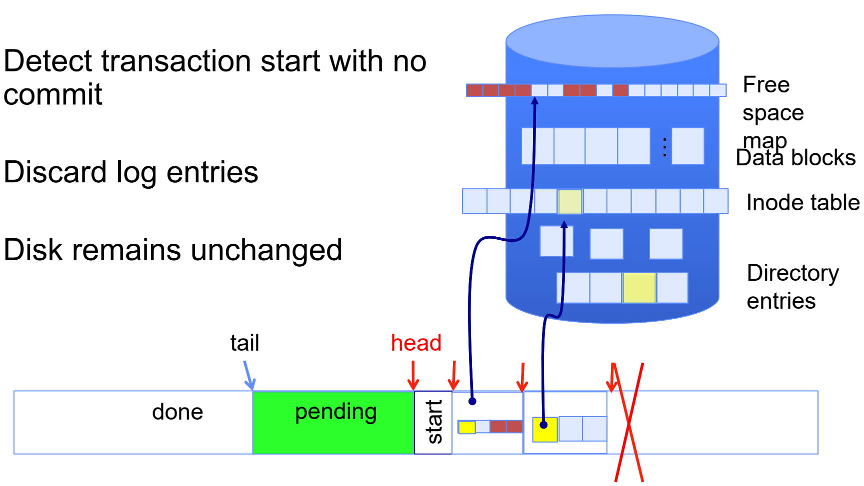
- After Commit
- All access to file system first looks in log
- Eventually copy changes to disk



Log: in non-volatile storage (Flash or Disk)

Crash During Logging – Recover

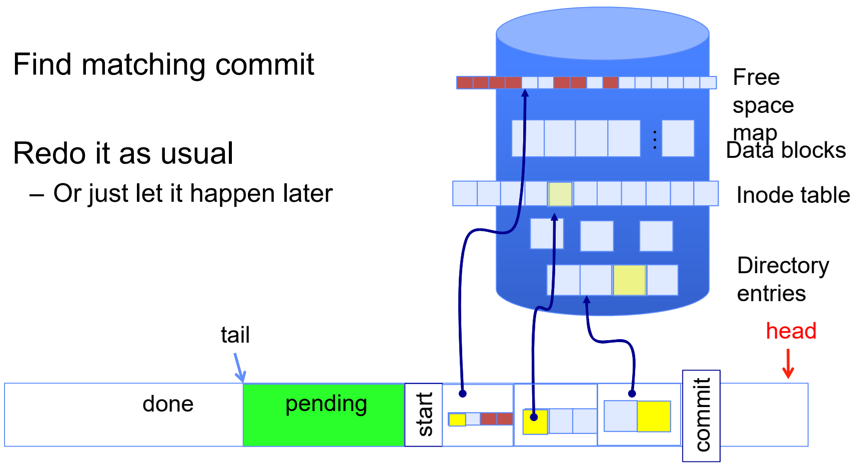
- Upon recovery scan the log
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



Log: in non-volatile storage (Flash or on Disk)

Recovery After Commit

- Scan log, find start
- Find matching commit
- Redo it as usual
  – Or just let it happen later



Log: in non-volatile storage (Flash or on Disk)

Journaling Summary

Why go through all this trouble?

- Updates atomic, even if we crash:
  – Update either gets fully applied or discarded
  – All physical operations *treated as a logical unit*

Isn't this expensive?

- Yes! We're now writing all data twice (once to log, once to actual data blocks in target file)
- Modern filesystems offer an option to journal metadata updates only
  – Record modifications to file system data structures
  – But apply updates to a file's contents directly

Going Further – Log Structured File Systems

- The log IS what is recorded on disk
  – File system operations *logically* replay log to get result
  – Create data structures to make this fast
  – On recovery, replay the log
- Index (inodes) and directories are written into the log too
- Large, important portion of the log is cached in memory
- Do everything in bulk: log is collection of large segments
- Each segment contains a summary of all the operations within the segment
  – Fast to determine if segment is relevant or not
- Free space is approached as continual cleaning process of segments
  – Detect what is live or not within a segment
  – Copy live portion to new segment being formed (replay)
  – Garbage collection entire segment
  – No bit map

LFS Paper in Readings

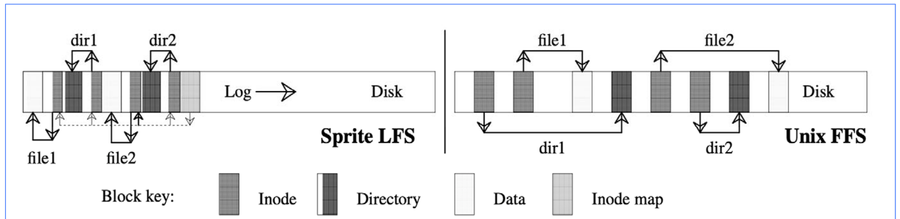


Figure 1 — A comparison between Sprite LFS and Unix FFS.

- LFS: write file1 block, write inode for file1, write directory page mapping "file1" in "dir1" to its inode, write inode for this directory page. Do the same for "/dir2/file2". Then write summary of the new inodes that got created in the segment
- FFS: <left as exercise>
- Reads are same in either case (pointer following)
- Buffer cache likely to hold information in both cases
  – But disk IOs are very different – writes sequential, reads not!
  – Randomness of read layout assumed to be handled by cache

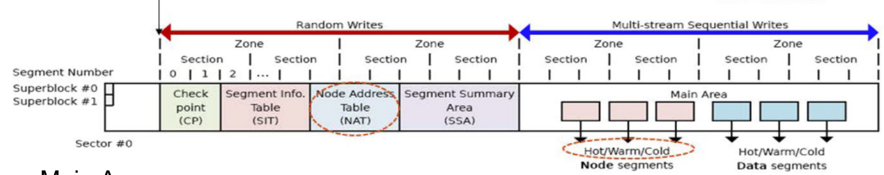
Example: F2FS: A Flash File System

- File system used on many mobile devices
  - Including the Pixel 3 from Google
  - Latest version supports block-encryption for security
  - Has been "mainstream" in linux for several years now
- Assumes standard SSD interface
  - With built-in Flash Translation Layer (FTL)
  - Random reads are as fast as sequential reads
  - Random writes are bad for flash storage
    - » Forces FTL to keep moving/coalescing pages and erasing blocks
    - » Sustained write performance degrades/lifetime reduced
- Minimize Writes/updates and otherwise keep writes "sequential"
  - Start with Log-structured file systems/copy-on-write file systems
  - Keep writes as sequential as possible
  - Node Translation Table (NAT) for "logical" to "physical" translation
    - » Independent of FTL
- For more details, check out paper in *Readings* section of website
  - "F2FS: A New File System for Flash Storage" (from 2015)
  - Design of file system to leverage and optimize NAND flash solutions
  - Comparison with Ext4, Btrfs, Nilfs2, etc

Flash-friendly on-disk Layout

Align to zone size

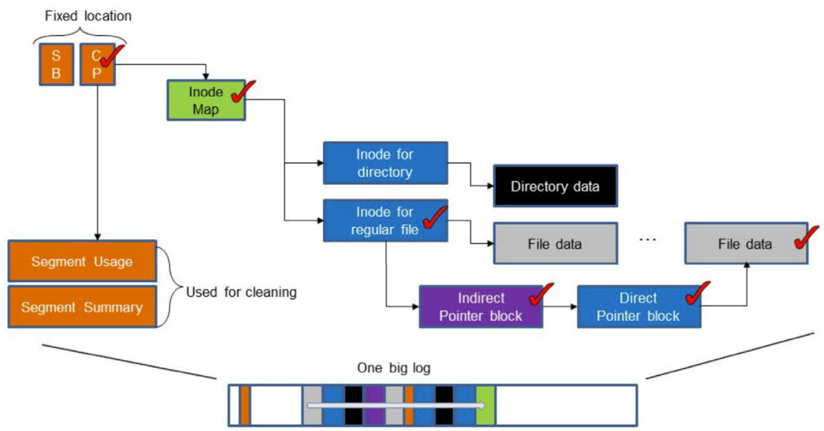
Block – 4KB
Segment – 2MB
Section – n Segments
Zone – m Sections



- Main Area:
  - Divided into segments (basic unit of management in F2FS)
  - 4KB Blocks. Each block typed to be *node* or *data*.
- Node Address Table (NAT): *Independent of FTL!*
  - Block address table to locate all "node blocks" in Main Area
- Updates to data sorted by predicted write frequency (Hot/Warm/Cold) to optimize FLASH management
- Checkpoint (CP): Keeps the file system status
  - Bitmaps for valid NAT/SIT sets and Lists of orphan inodes
  - Stores a consistent F2FS status at a given point in time
- Segment Information Table (SIT):
  - Per segment information such as number of valid blocks and the bitmap for the validity of all blocks in the "Main" area
  - Segments used for "garbage collection"
- Segment Summary Area (SSA):
  - Summary representing the owner information of all blocks in the Main area

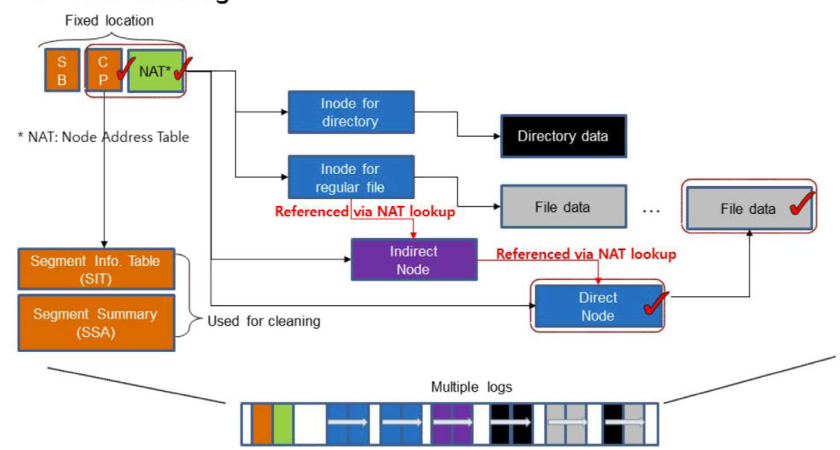
LFS Index Structure: Forces many updates when updating data

- **Update propagation issue: wandering tree**
- **One big log**



F2FS Index Structure: Indirection and Multi-head logs optimize updates

- **Restrained update propagation: node address translation method**
- **Multi-head log**



File System Summary (1/3)

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for size, access and usage patterns
  - Maximize sequential access, allow efficient random access
  - Projects the OS protection and security regime (UGO vs ACL)
- File defined by header, called "inode"
- Naming: translating from user-visible names to actual sys resources
  - Directories used for naming for local file systems
  - Linked or tree structure stored in files
- Multilevel Indexed Scheme
  - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc..
  - NTFS: variable extents not fixed blocks, tiny files data is in header

File System Summary (2/3)

- File layout driven by freespace management
  - Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization
  - Integrate freespace, inode table, file blocks and dirs into block group
- FLASH filesystems optimized for:
  - Fast random reads
  - Limiting Updates to data blocks
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks yet on disk)

File System Summary (3/3)

- File system operations involve multiple distinct updates to blocks on disk
  - Need to have all or nothing semantics
  - Crash may occur in the midst of the sequence
- Traditional file system perform check and recovery on boot
  - Along with careful ordering so partial operations result in loose fragments, rather than loss
- Copy-on-write provides richer function (versions) with much simpler recovery
  - Little performance impact since sequential write to storage device is nearly free
- Transactions over a log provide a general solution
  - Commit sequence to durable log, then update the disk
  - Log takes precedence over disk
  - Replay committed transactions, discard partials