CS162
Operating Systems and Systems Programming
Lecture 19

Buffering and Reliability in File Systems

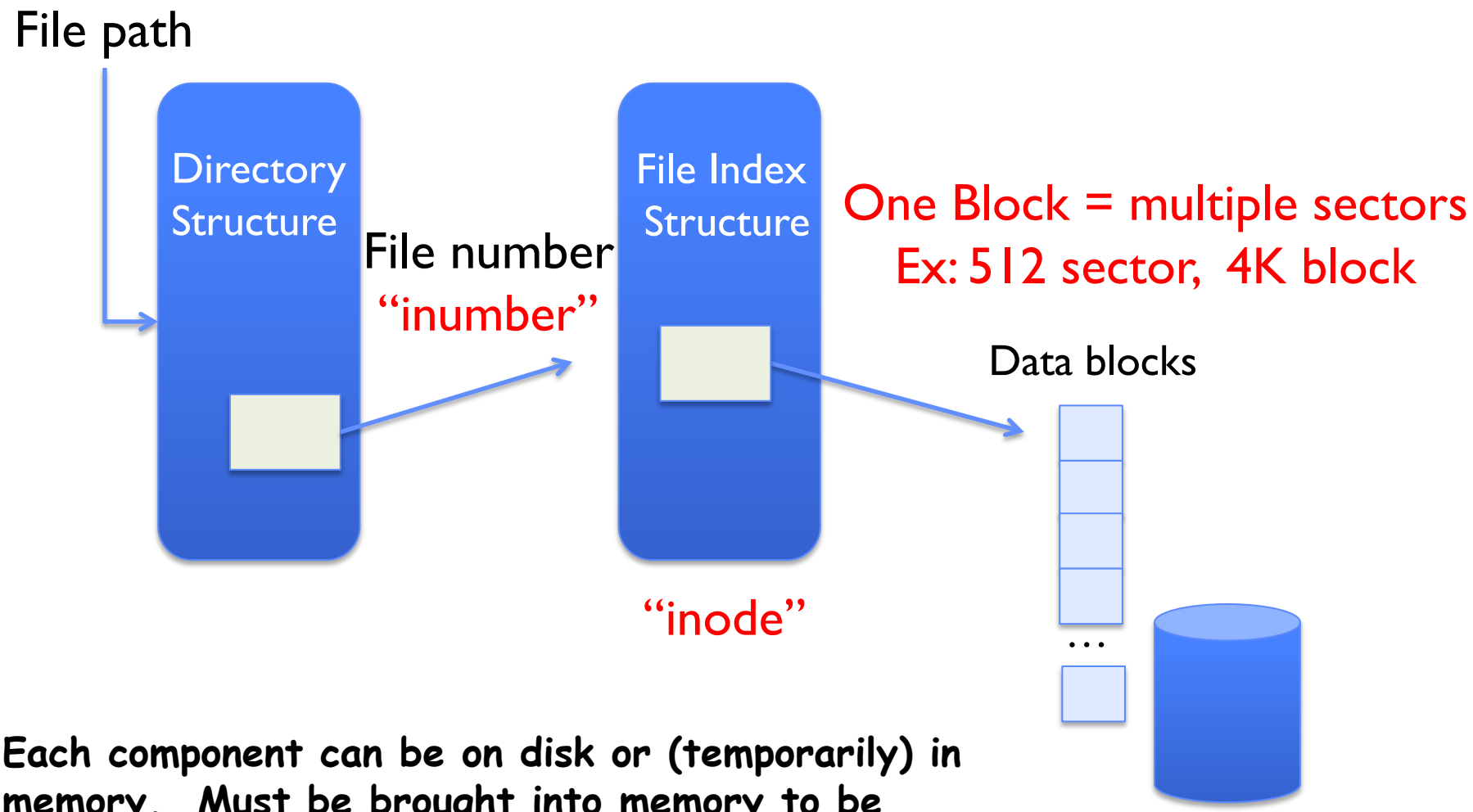
November 5, 2019

Prof. David E. Culler

http://cs162.eecs.Berkeley.edu

Read: A&D Ch 14

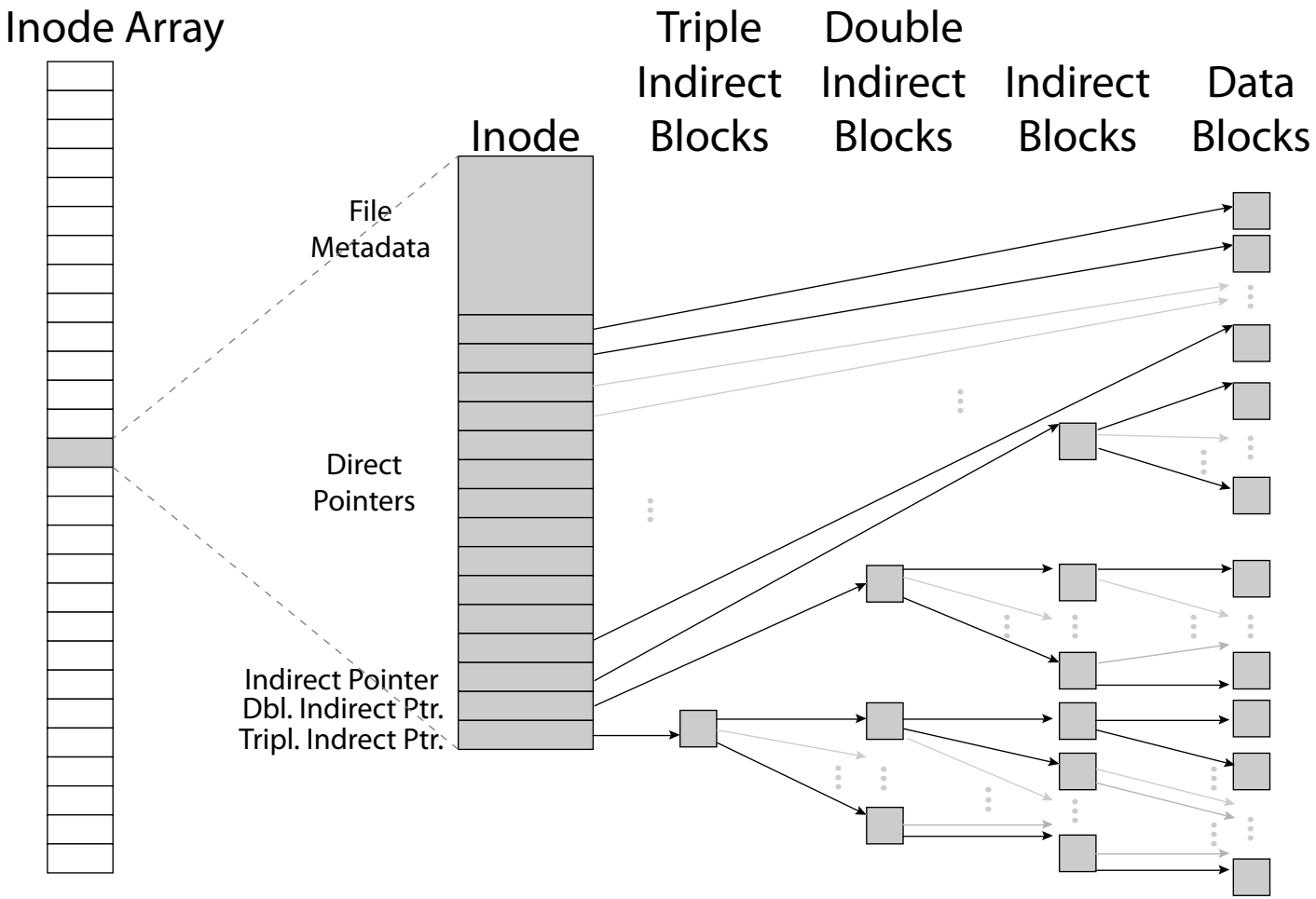
Recall: Components of a File System



Each component can be on disk or (temporarily) in memory. Must be brought into memory to be accessed.

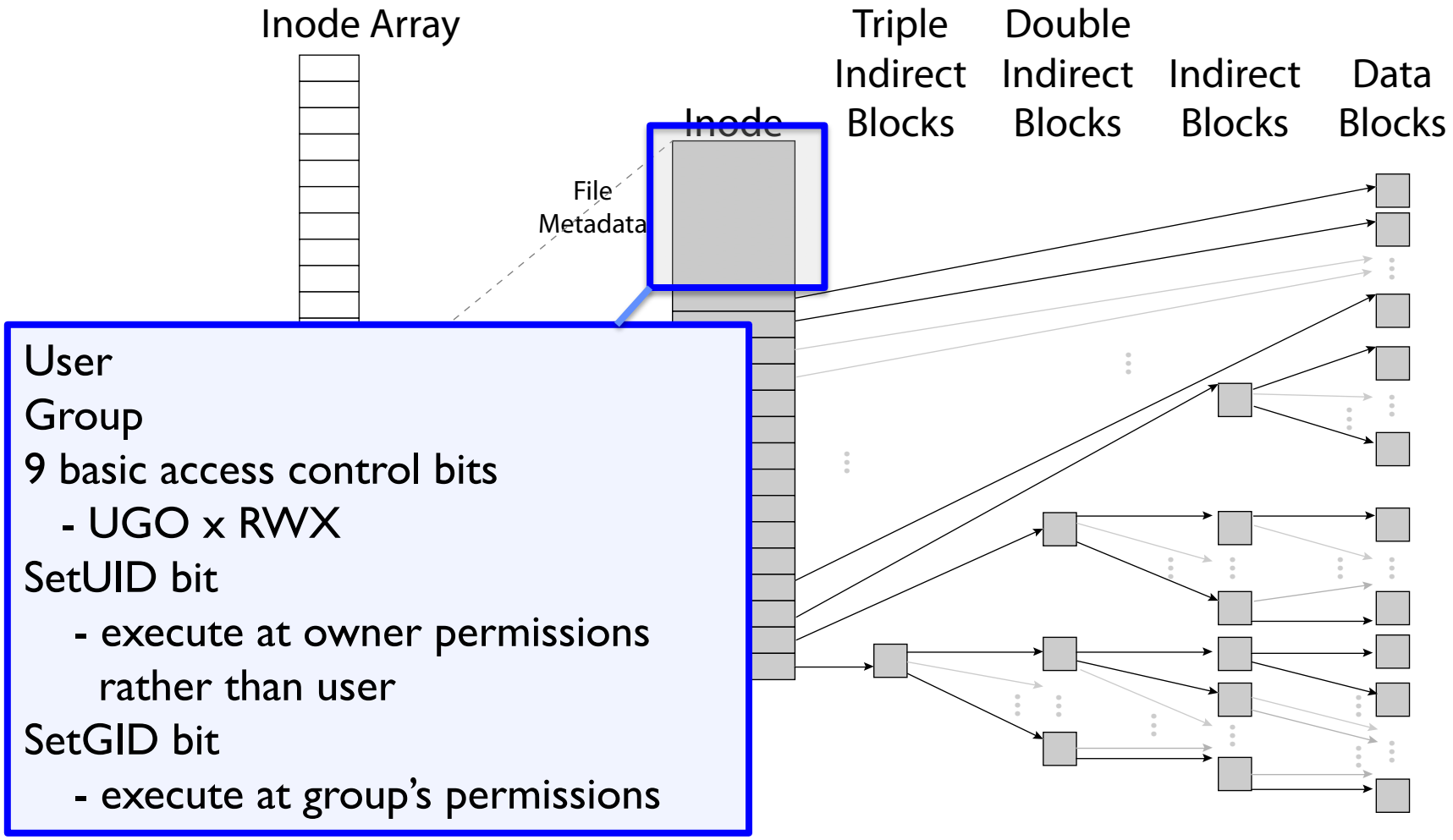
Inode Structure

- inode metadata



File Attributes

- inode metadata



Data Storage

- Small files: 12 pointers direct to data blocks

Direct pointers

4kB blocks ⇒ sufficient for files up to 48KB

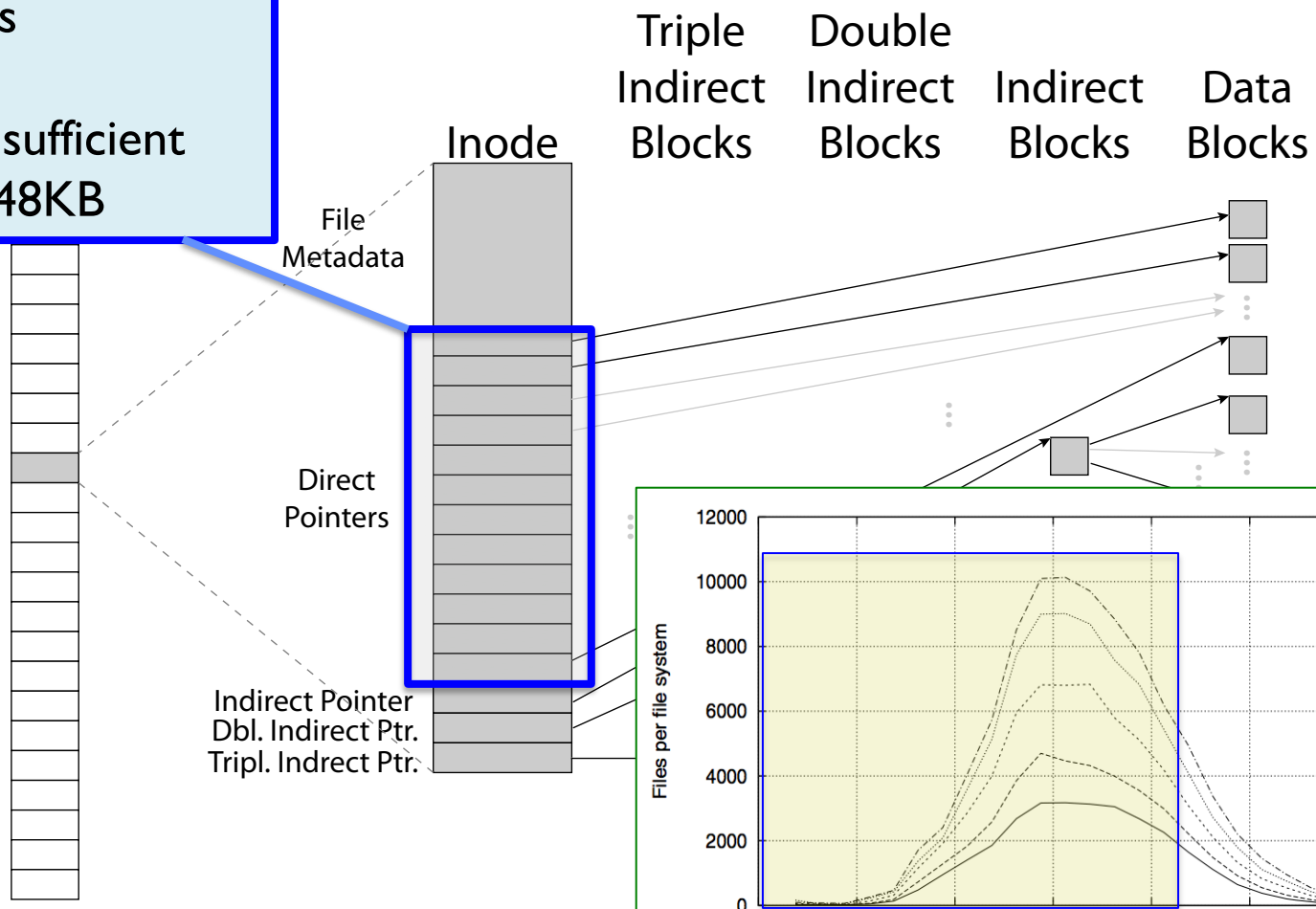
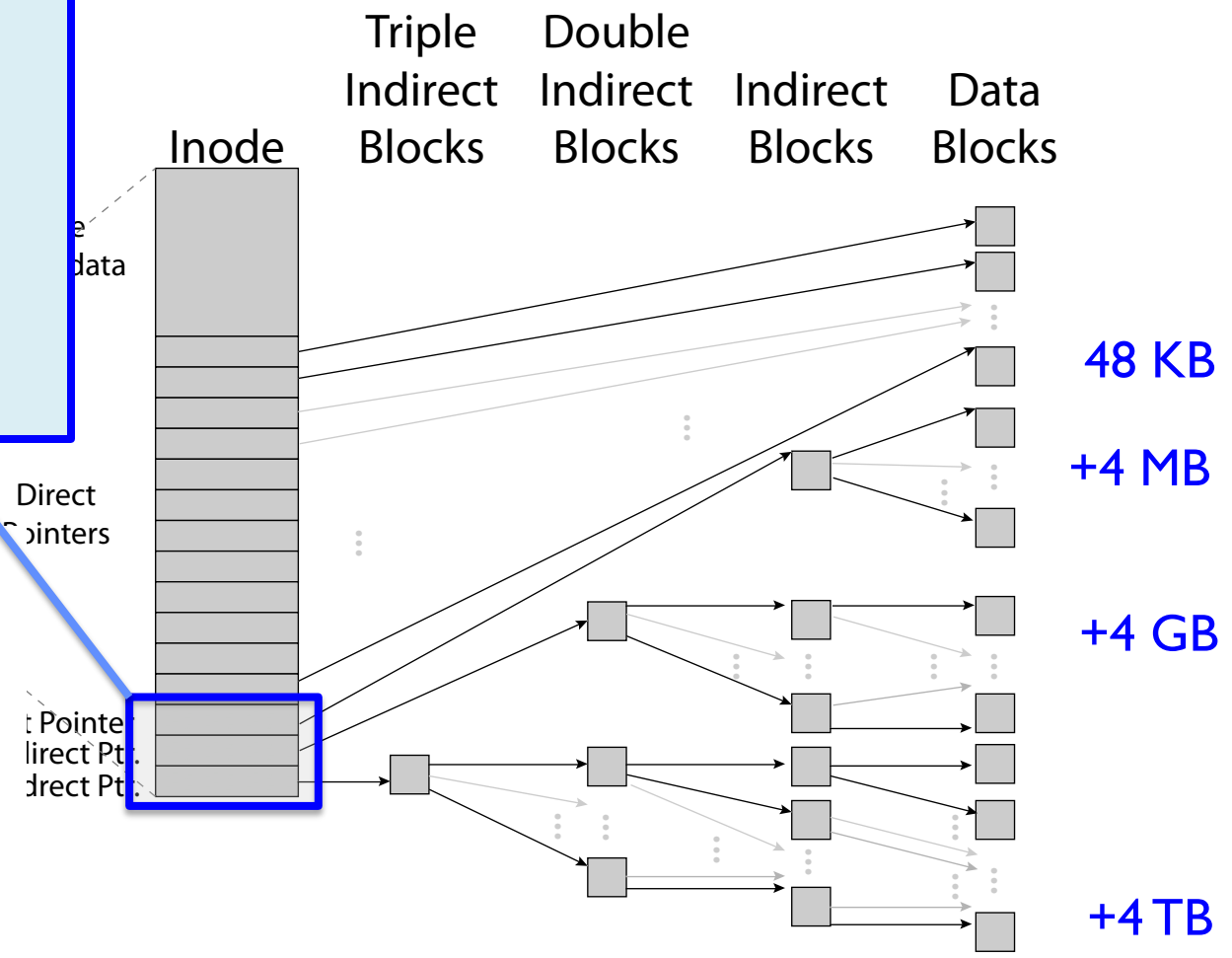


Fig. 2. Histograms of files by size.

Data Storage

- Large files: 1,2,3 level indirect pointers

Indirect pointers
- point to a disk block containing only pointers
- 4 kB blocks => 1024 ptrs
  => 4 MB @ level 2
  => 4 GB @ level 3
  => 4 TB @ level 4



A Five-Year Study of File-System Metadata • 9:9

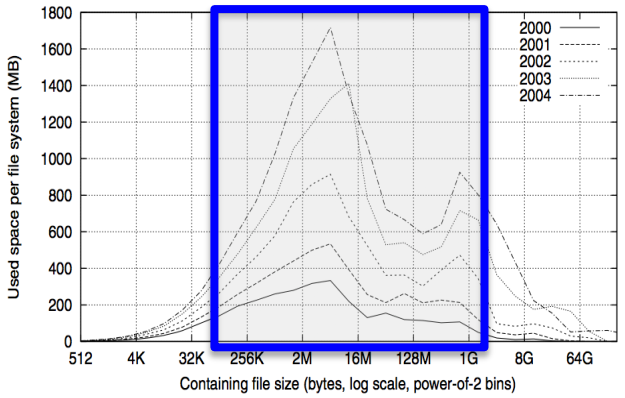
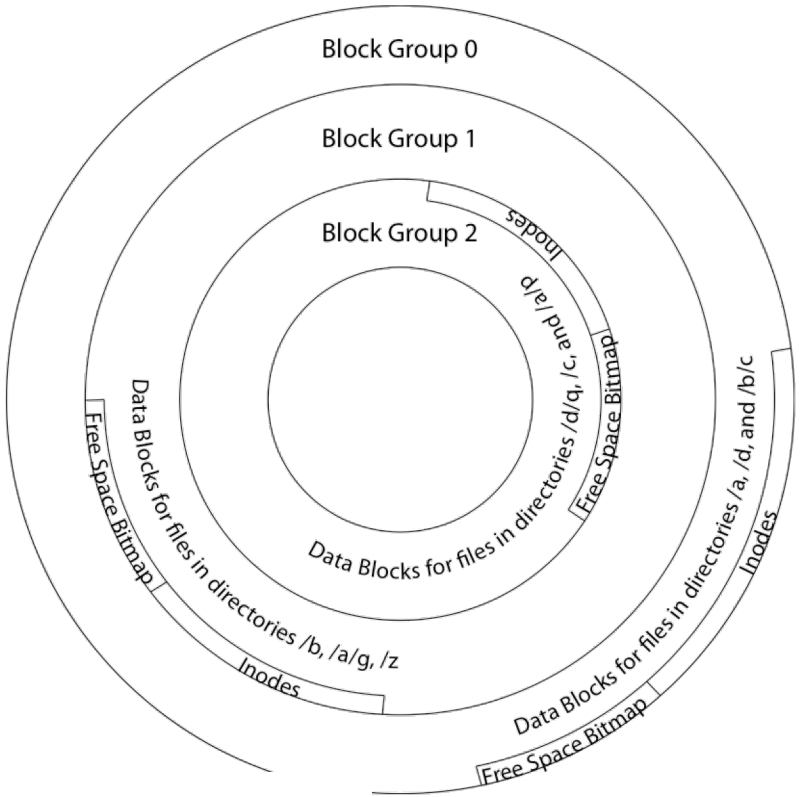
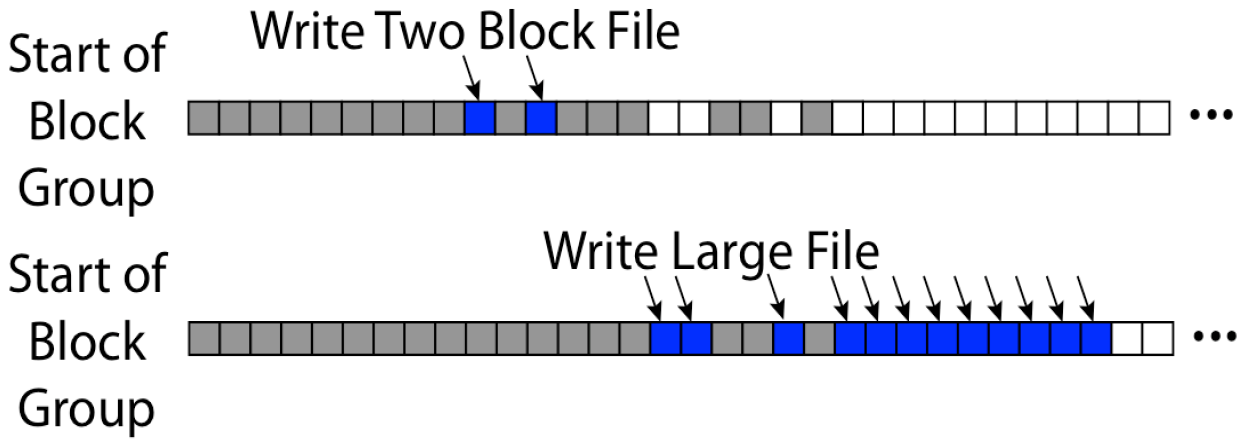


Fig. 4. Histograms of bytes by containing file size.

4.2 BSD Locality: Block Groups

- First-Free allocation of new file blocks
  - To expand file, first try successive blocks in bitmap, then choose new range of blocks
  - Few little holes at start, big sequential runs at end of group
  - Avoids fragmentation
  - Sequential layout for big files
- Important: keep 10% or more free!
  - Reserve space in the Block Group

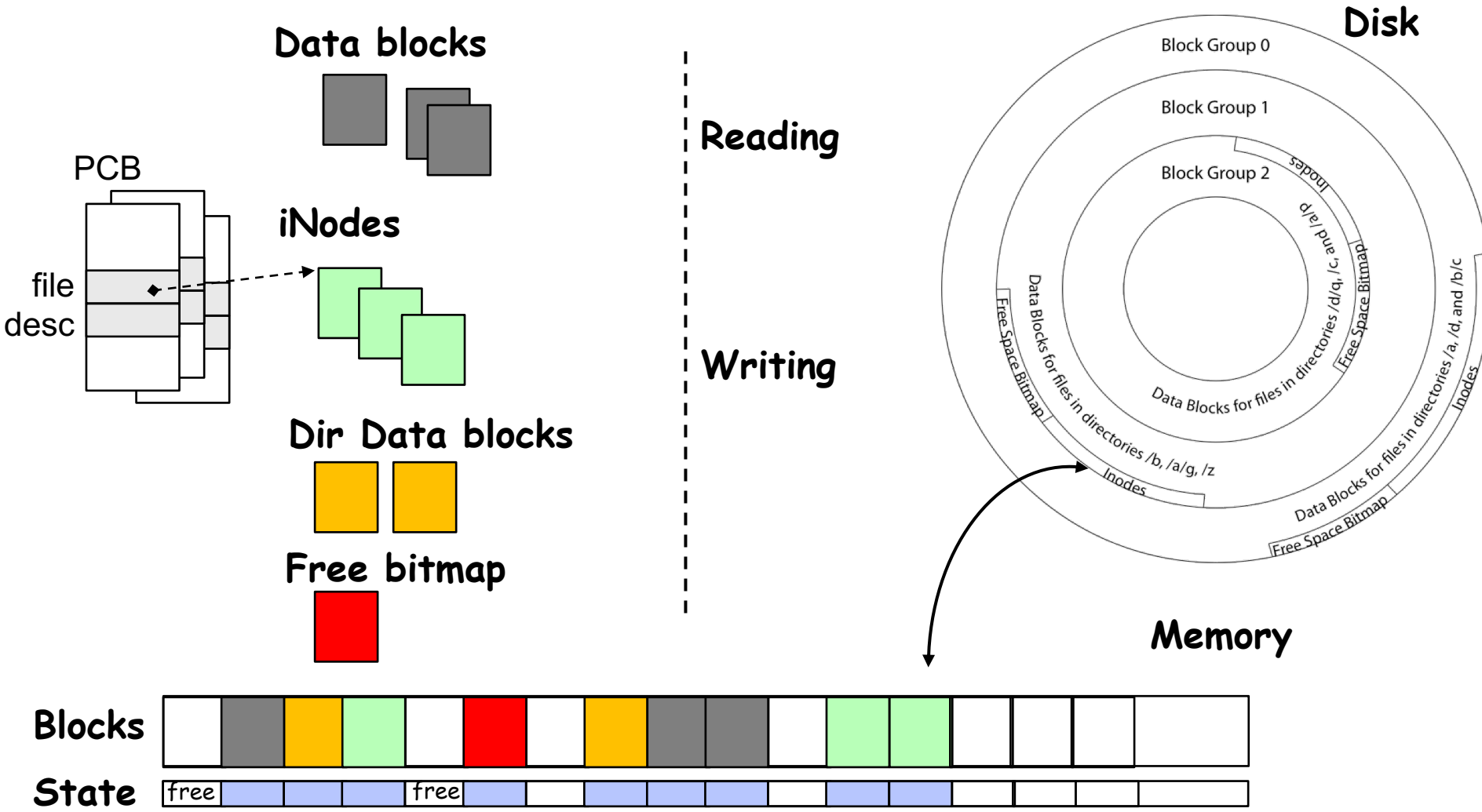




Buffer Cache

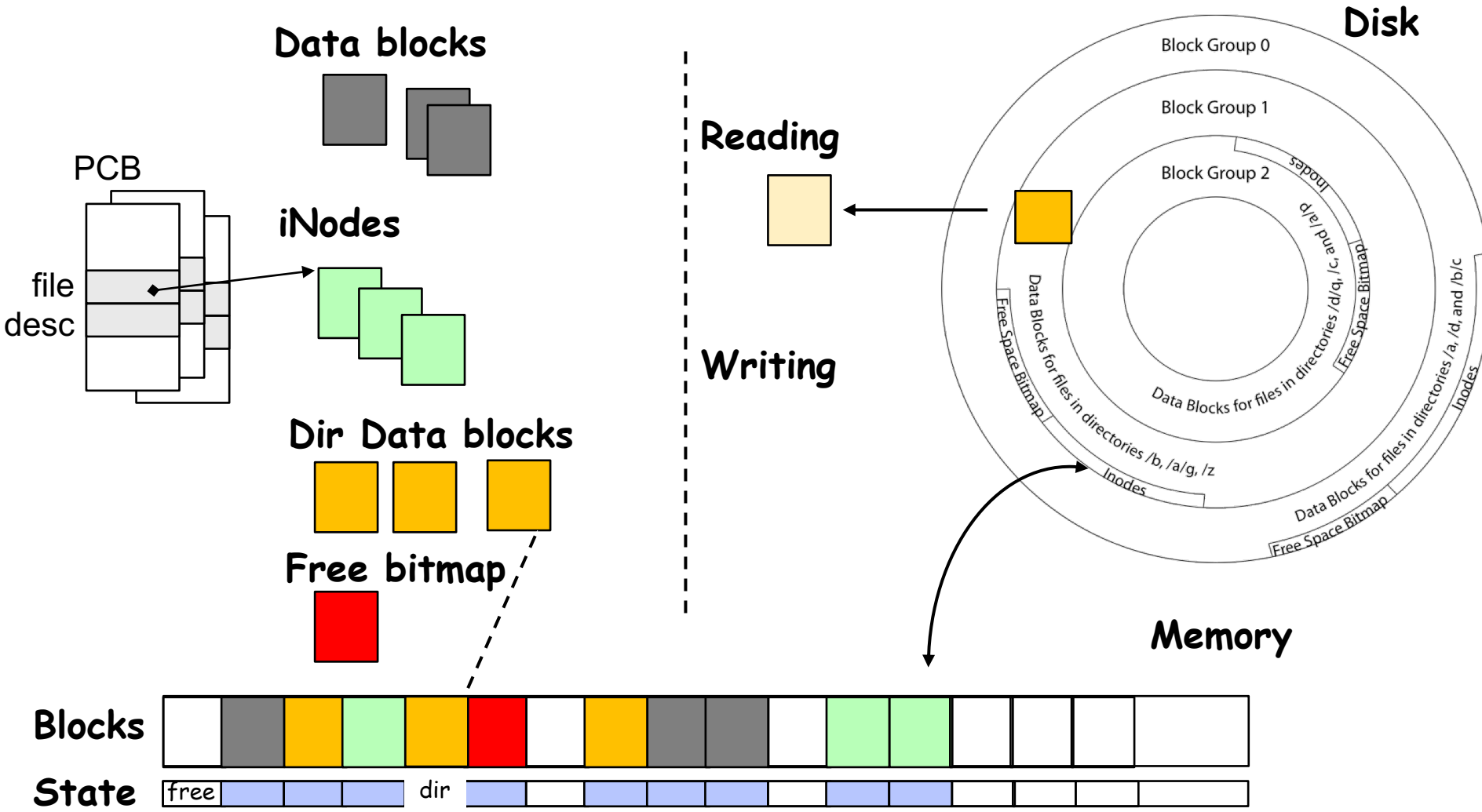
- Kernel *must* copy disk blocks to main memory to access their contents and write them back if modified
  - Could be data blocks, inodes, directory contents, etc.
  - Possibly dirty (modified and not written back)
- Key Idea: Exploit locality by caching disk data in memory
  - Name translations: Mapping 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



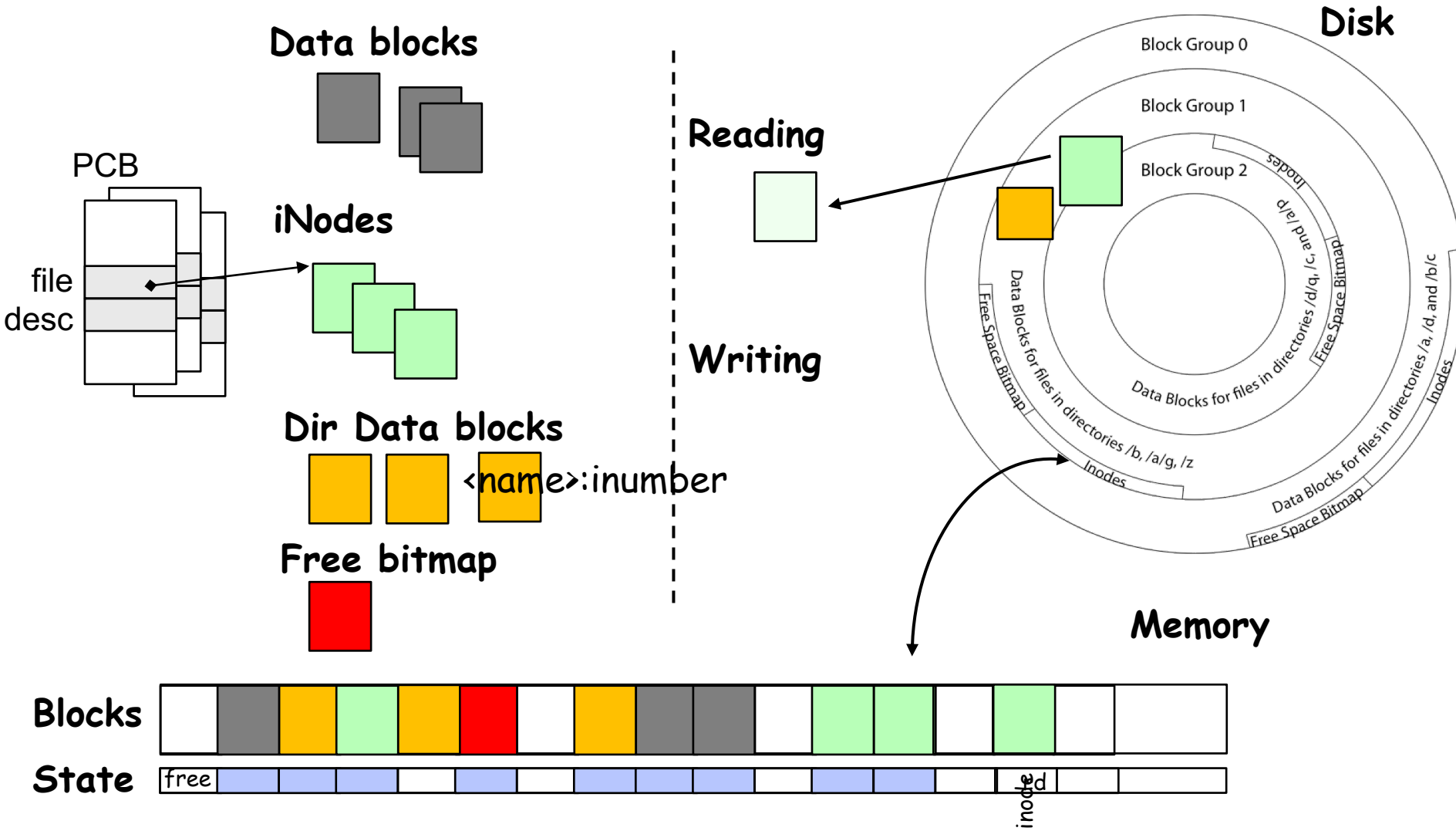
- OS implements a cache of disk blocks for efficient access to data, directories, inodes, freemap

File System Buffer Cache: open



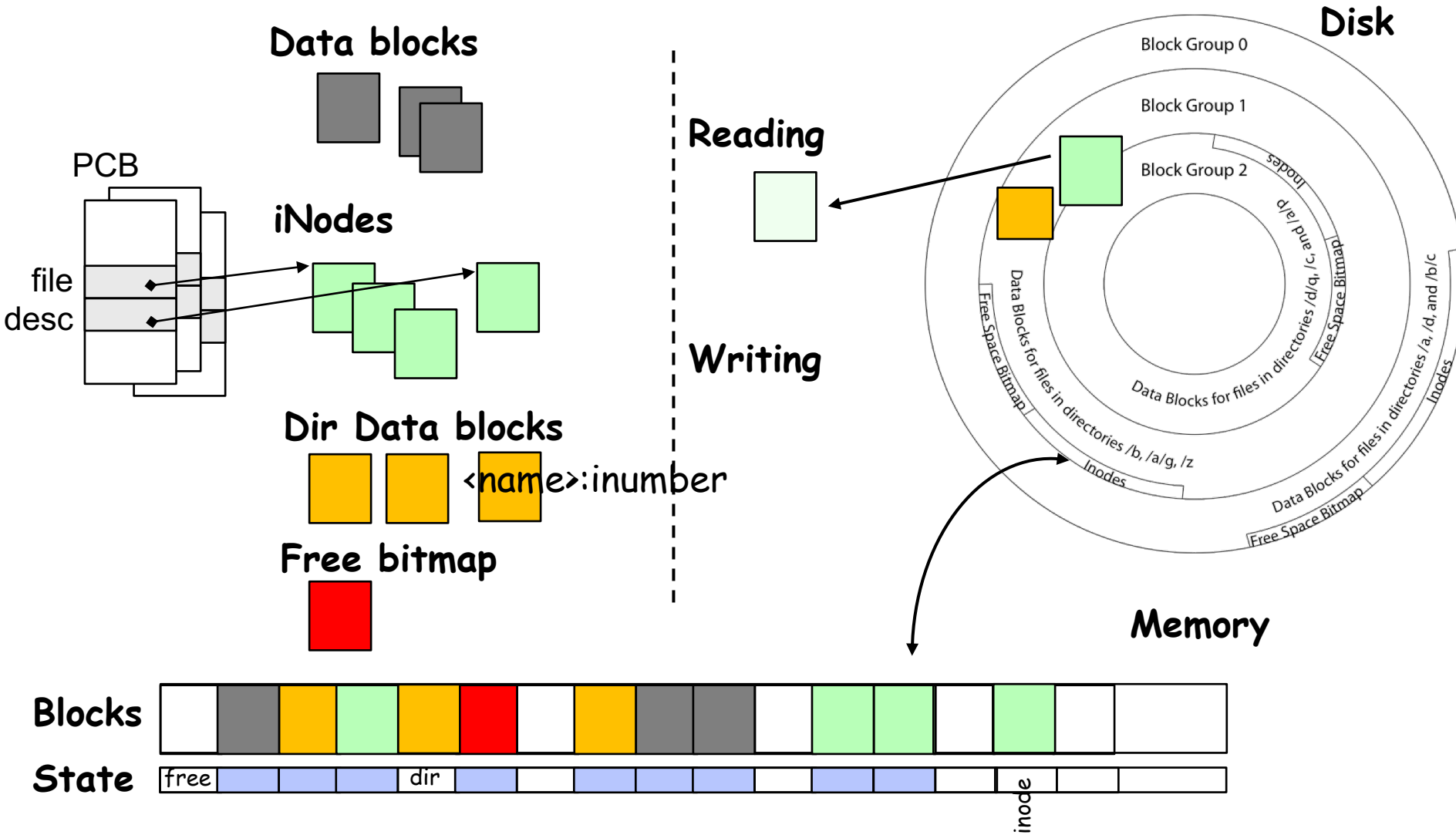
- {load block of directory; search for map}+ ;

File System Buffer Cache: open



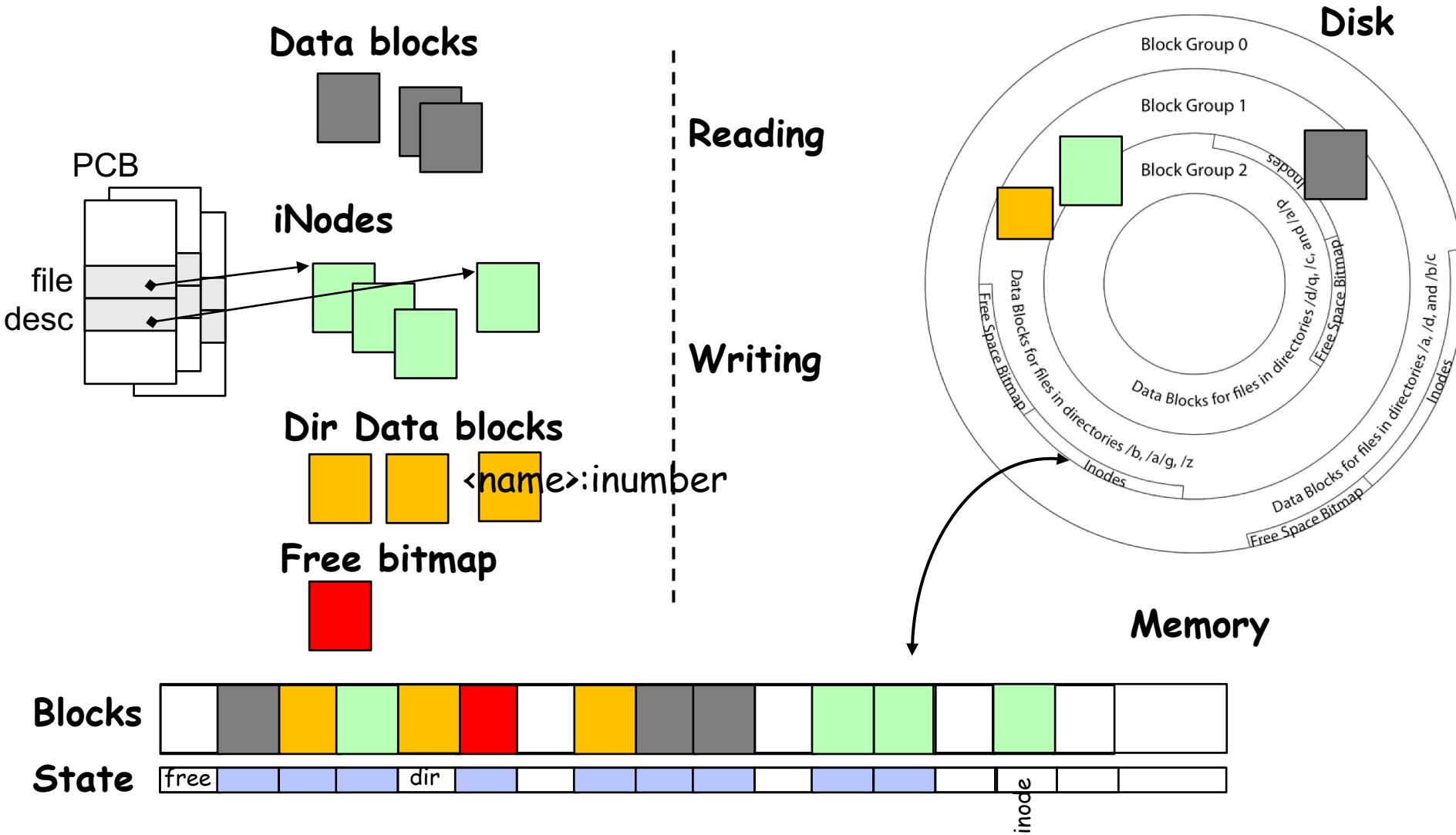
- {load block of directory; search for map}+ ; Load inode ;

File System Buffer Cache: open



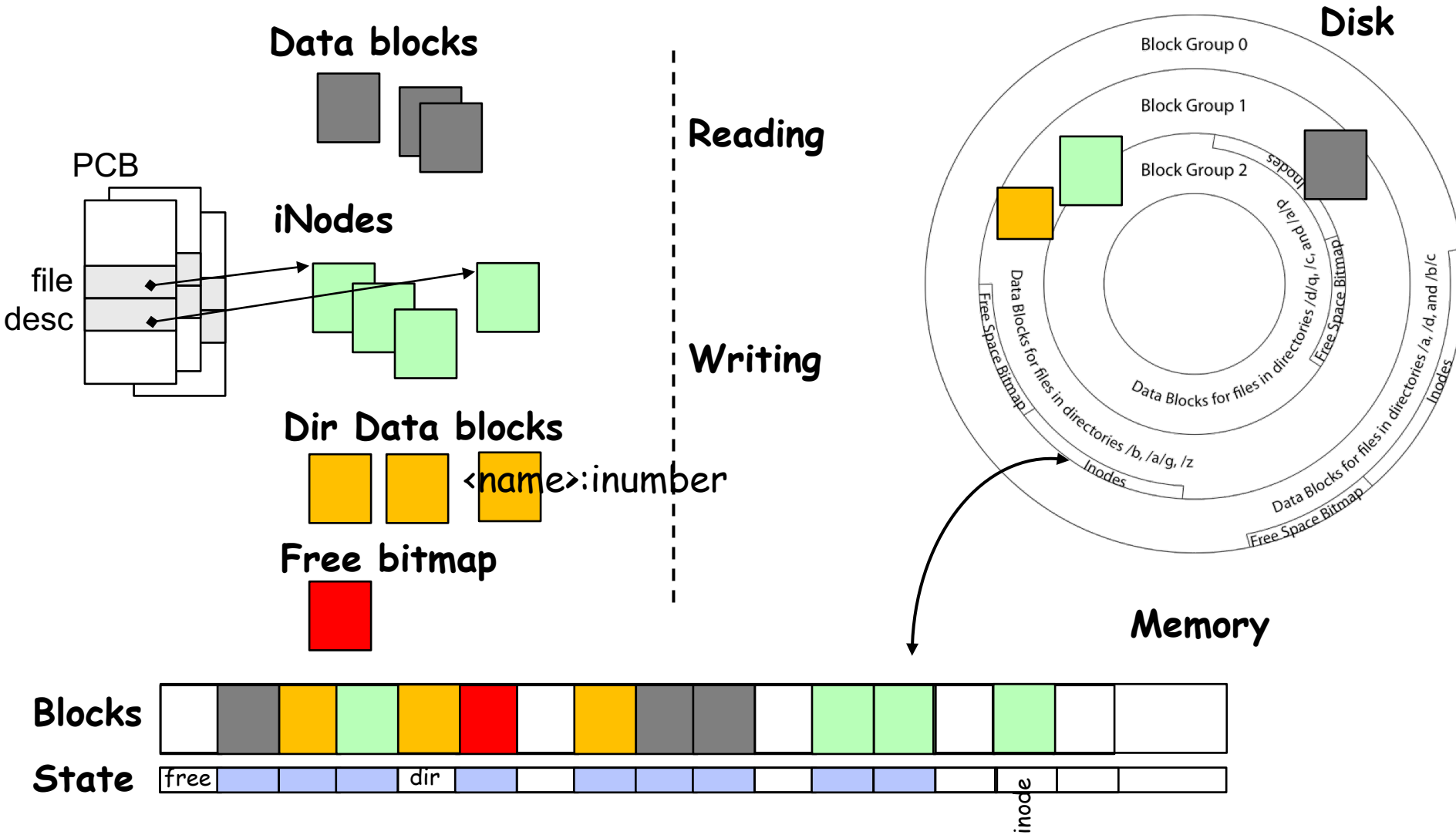
- {load block of directory; search for map}+ ; Load inode ;
- Create reference via open file descriptor

File System Buffer Cache: Read?



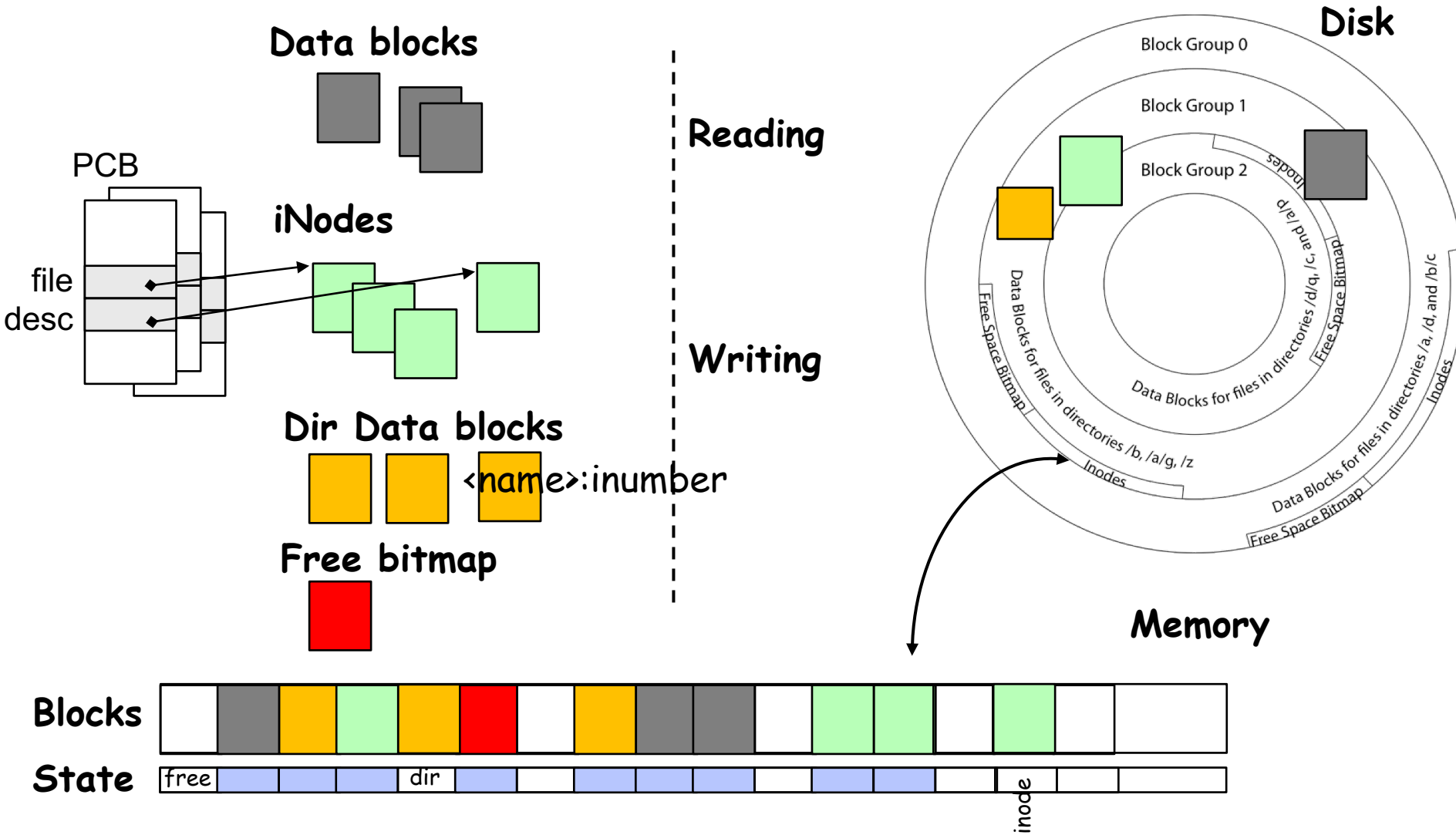
- From inode, traverse index structure to find data block; load data block; copy all or part to read data buffer

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

Issues to Implementing a Buffer Cache

- 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 of timestamps for each disk block
  - 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 file blocks.
  - Challenges:
    - » 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

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
- Read Ahead Prefetching: fetch sequential blocks early
  - Fast to access; File System tries to obtain sequential layout; Applications tend to do sequential reads and writes
- How much to prefetch?
  - » Too many delays on requests by other applications
  - » Too few causes many seeks (and rotational delays) among concurrent file requests
- "Elevator algorithm" can efficiently interleave groups of prefetches from concurrent applications

Delayed Writes

- Writes not immediately sent to disk
  - So buffer cache is a write back cache
- **`write`** copies data from user space to kernel buffer
  - 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

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

Recall: Important "ilities"

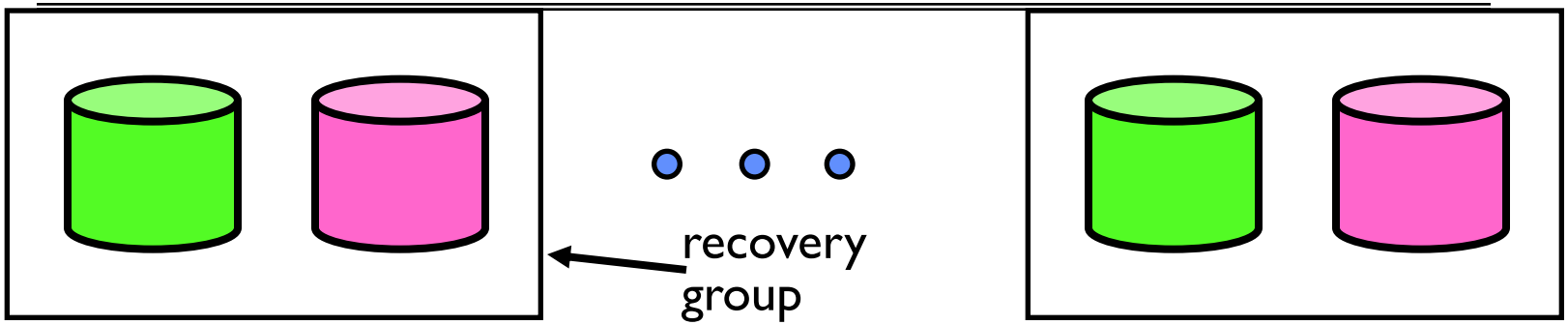
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time
  - the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems
- Availability: the probability that the system can accept and process requests
  - Can build highly-available systems, despite unreliable components
    - » Involves independence of failures and redundancy
  - Often as "nines" of probability. 99.9% is "3-nines of availability"
- 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

Break

Recall: 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…

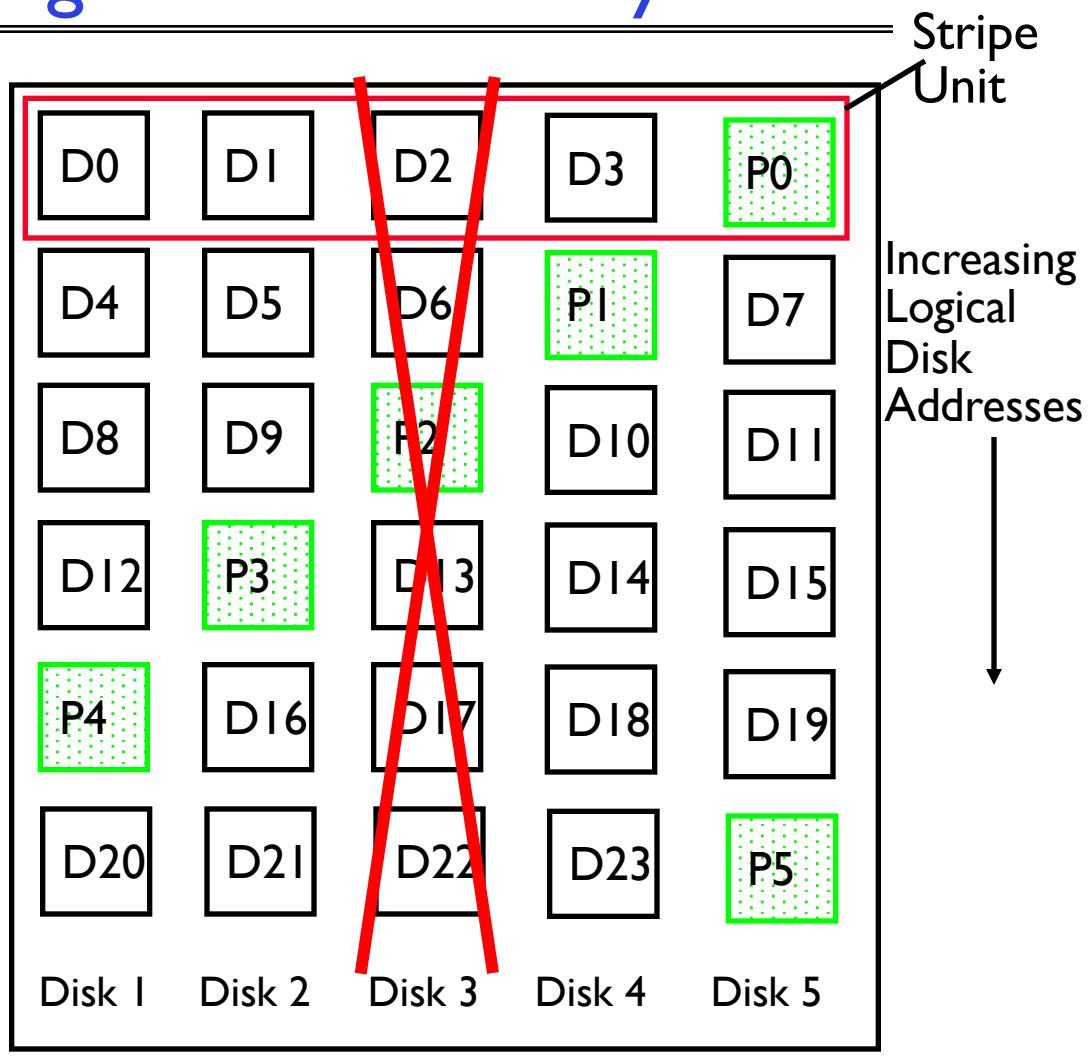
RAID 1: Disk Mirroring/Shadowing



- 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⊕D1⊕D2⊕D3
  - Can destroy any one disk and still reconstruct data
  - Suppose Disk 3 fails, then can reconstruct: D2=D0⊕D1⊕D3⊕P0



- Can spread information more widely for durability
  - RAID algorithms (and generalizations) within data center across cloud

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!

Threats to Reliability

- 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
- Interrupted Operation
  - Crash or power failure in the middle of a series of related updates may leave stored data in an inconsistent state
- Loss of stored data
  - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

2 Reliability Approaches

- Careful Ordering & Recovery
  - FAT & FFS + (fsck)
  - Each step builds structure,
    - » Data block << inode << free << directory
  - last links it in
  - Recover scans structure looking for incomplete actions
- Versioning
  - ZFS, OpenZFS, WAFL
  - At some granularity…
  - Create new structure linking back to unchanged parts of old
  - Last step is to declare new version exists

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

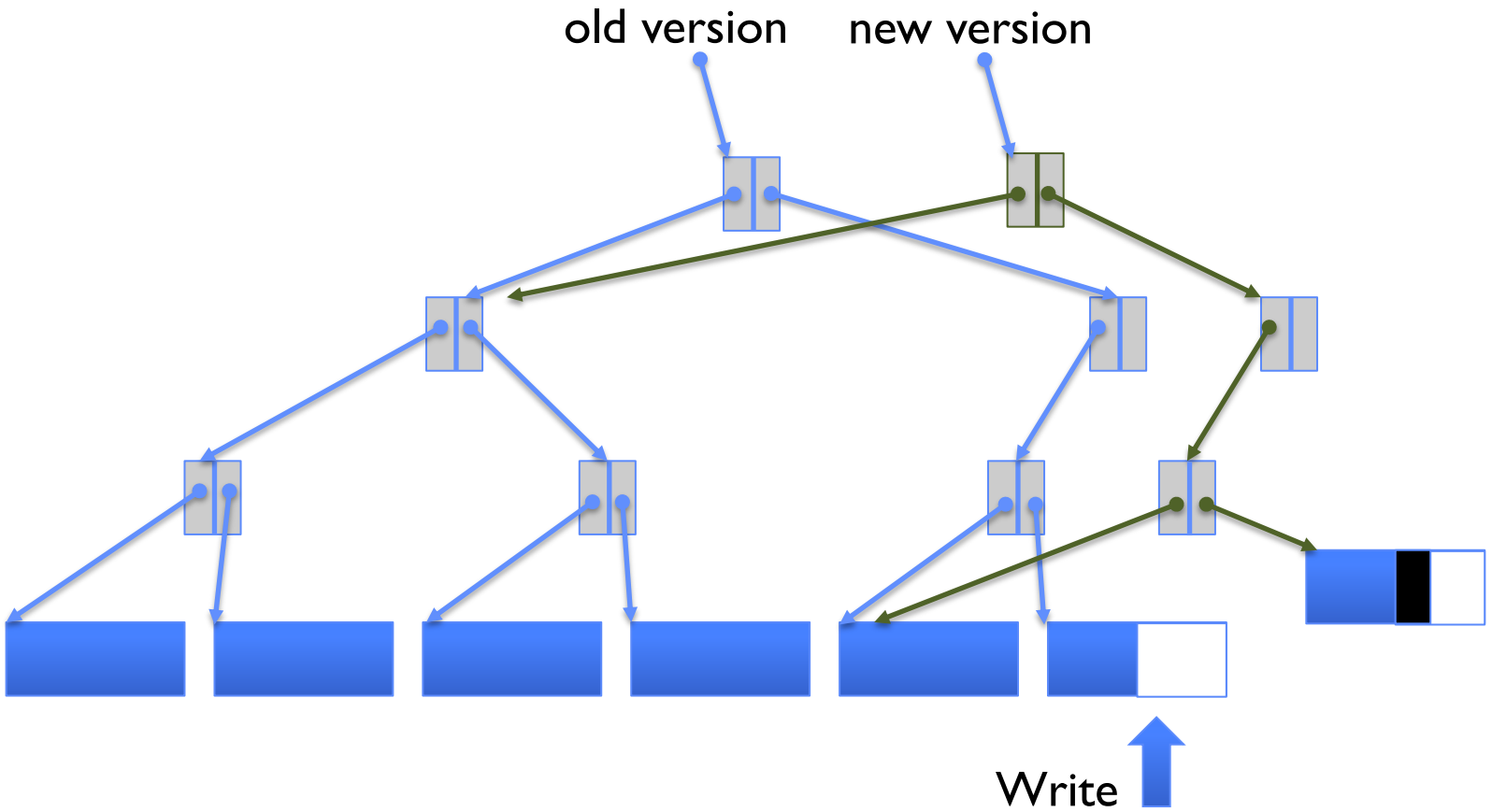
Time proportional to disk size

From Indexing to Versioning

- We need a multi-level index structure to find all the data blocks of a file
- And to allow it to grow as it is written
- Instead of over-writing existing data blocks and updating the index structure …
- Create a new version of the file with the updated data
- Reusing much of what is already in place

=> Copy On Write

COW with Smaller-Radix Blocks



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

More General Approach

- Use *transactions* for atomic updates
  - Ensure that multiple related operations performed atomically
  - If a crash occurs in middle, state of system should reflect all or none of the operations

- Most modern file systems use transactions to safely update their internals

Key Concept: Transaction

- Closely related to critical sections for manipulating shared data structures

- Extend concept of an atomic update from *memory* to *persistent storage*
  - Atomically update ***multiple*** persistent data structures

Key Concept: Transaction

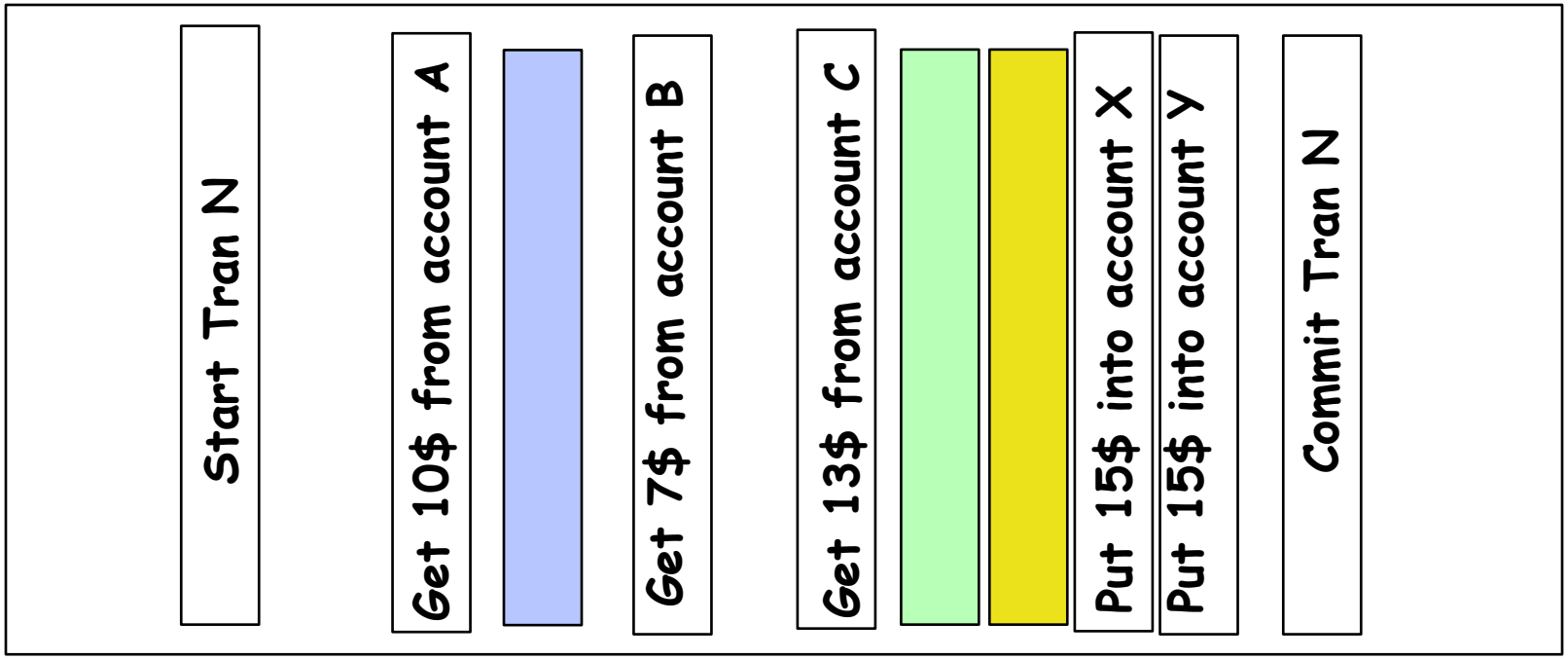
- Defined as an *atomic sequence* of reads/writes
- Takes system from one consistent state to another

Typical Transaction 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

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



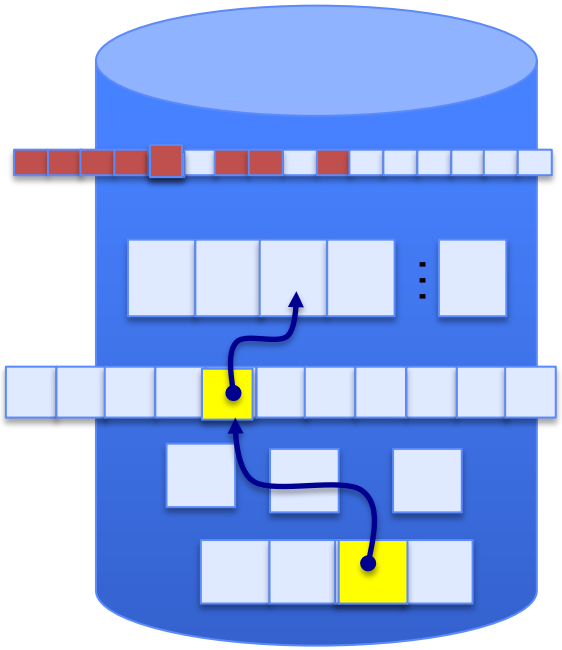
Journaling File Systems

- Don't modify data structures on disk directly
- Write each update as transaction recorded in a log
  - Commonly called a *journal* or *intention list*
  - Also maintained on disk (allocate blocks for it when formatting)
- Once changes are in the log, they can be safely applied
  - e.g. modify inode pointers and directory mapping
- Garbage collection: once a change is applied, remove its entry from the log

Example: Creating a File

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

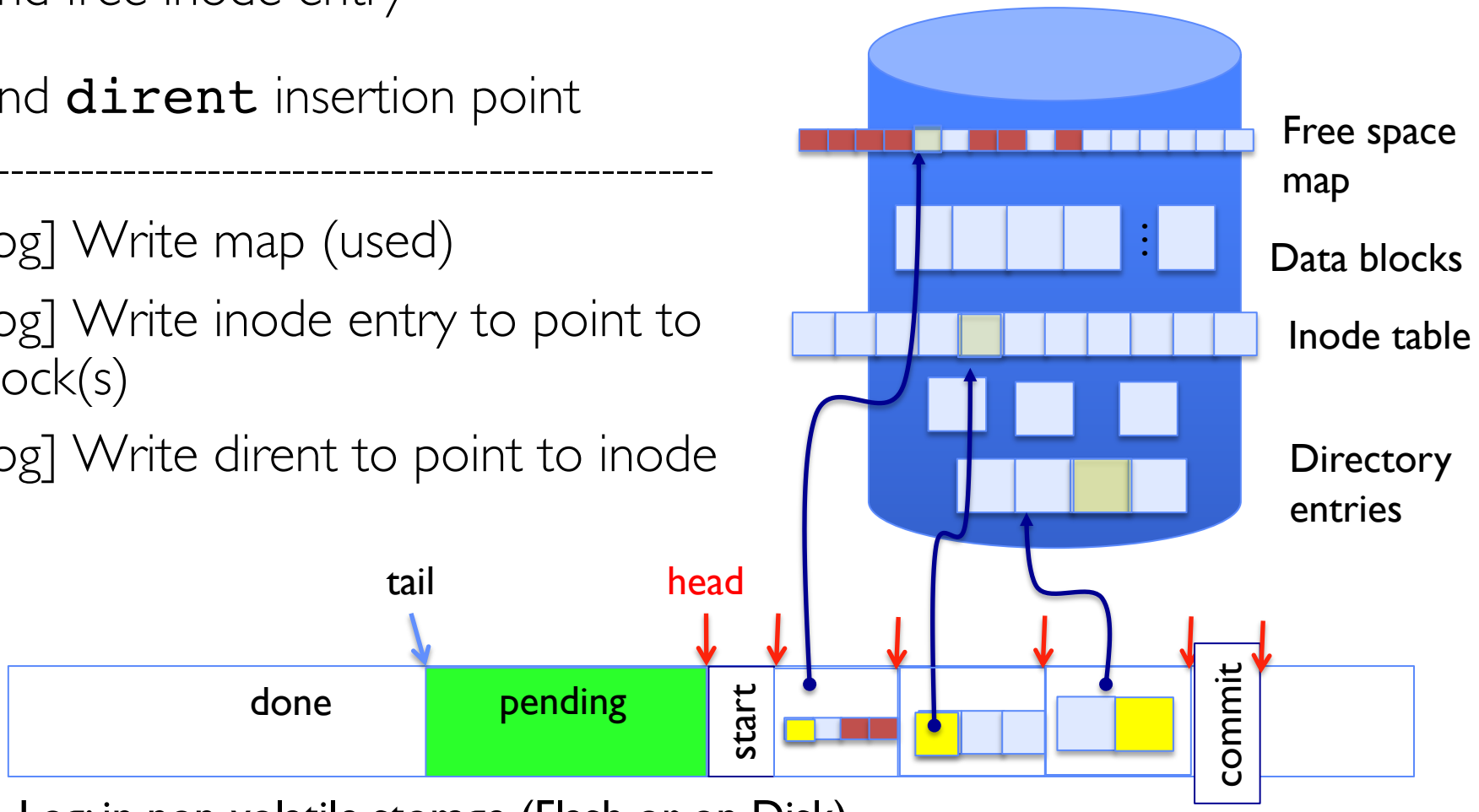
- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write **`dirent`** to point to inode



Ex: Creating a file (as transaction)

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

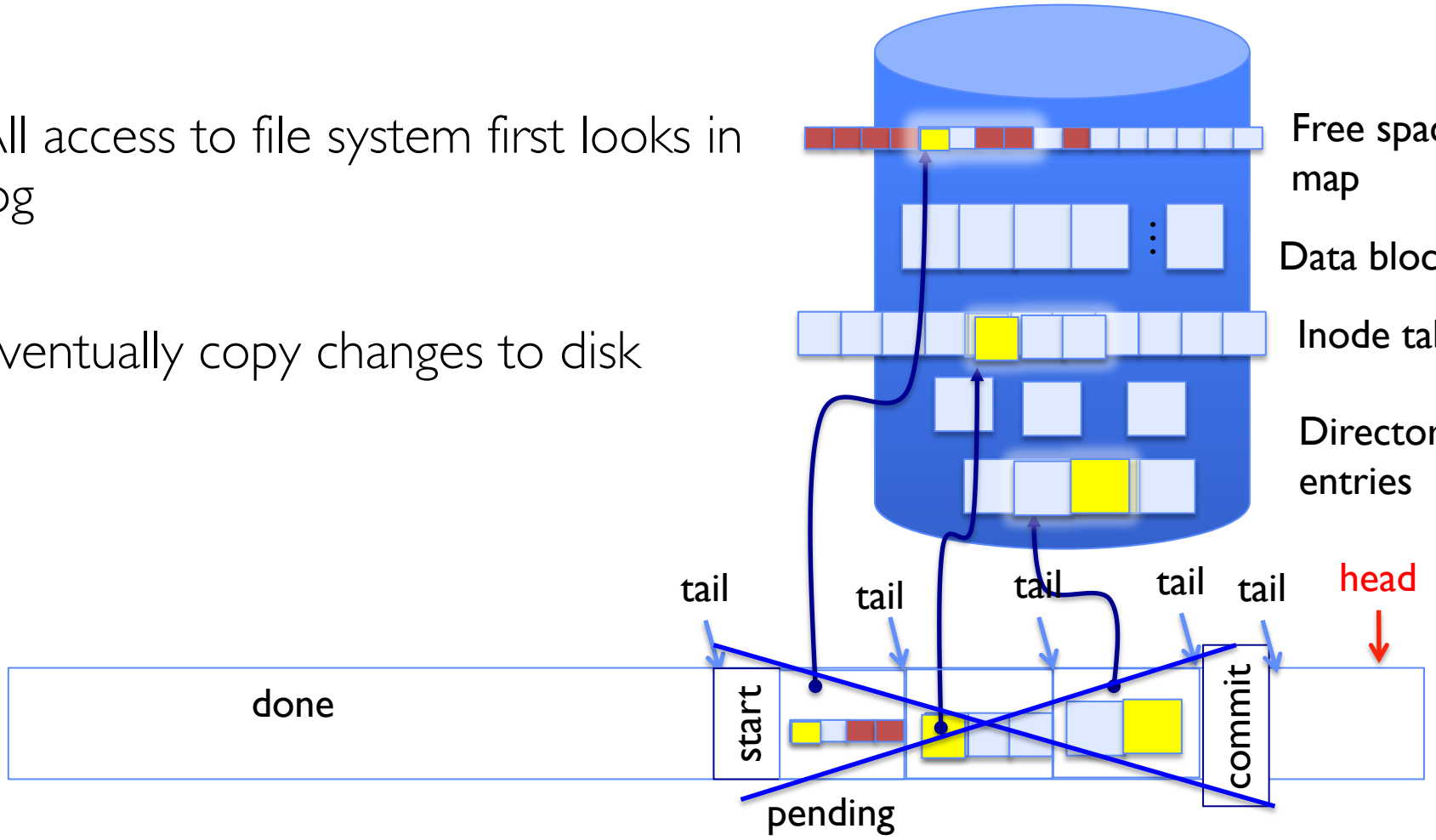
- [log] Write map (used)
- [log] Write inode entry to point to block(s)
- [log] Write dirent to point to inode



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

"Redo Log" – Replay Transactions

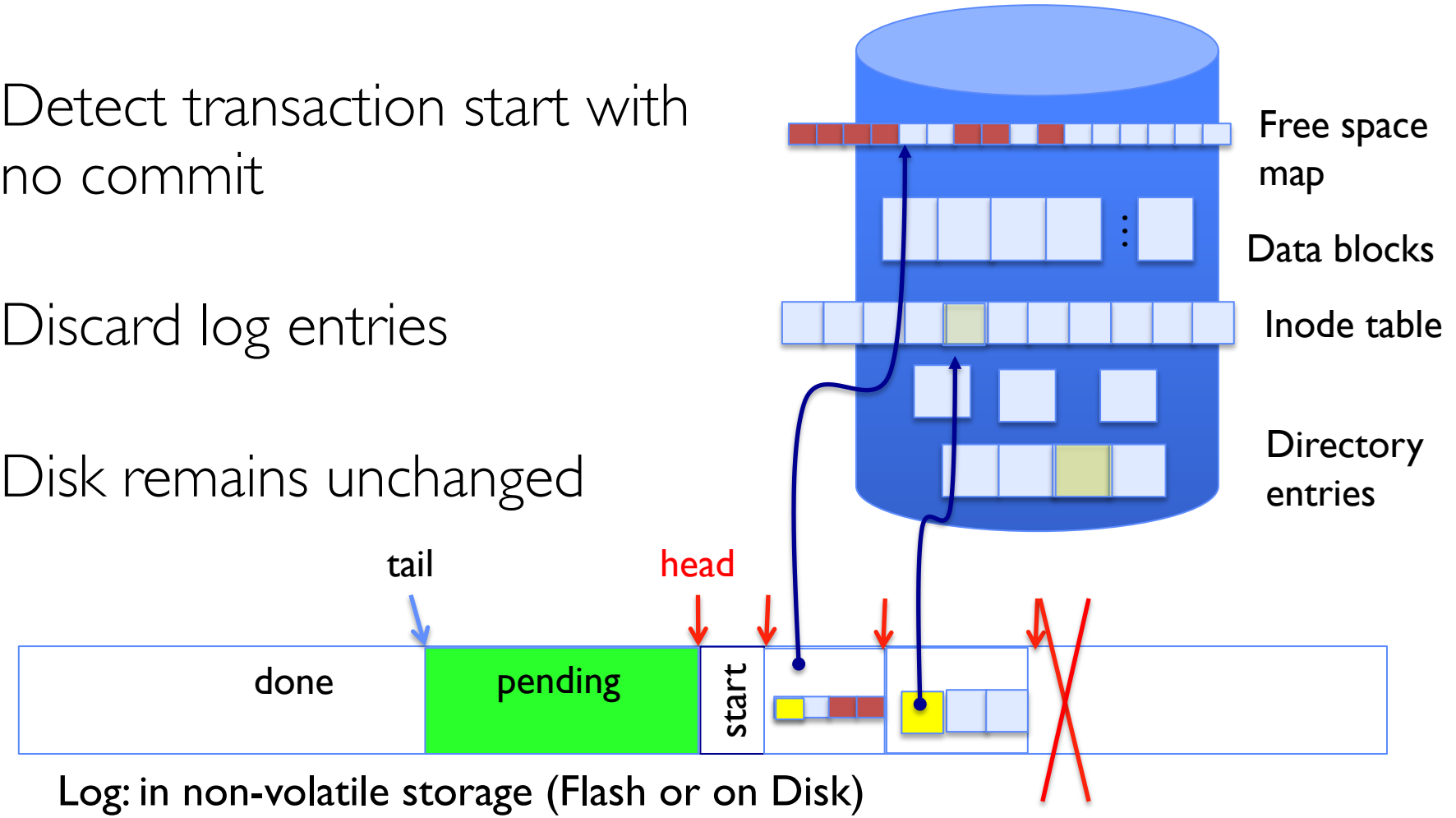
- After Commit
- All access to file system first looks in log
- Eventually copy changes to disk



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

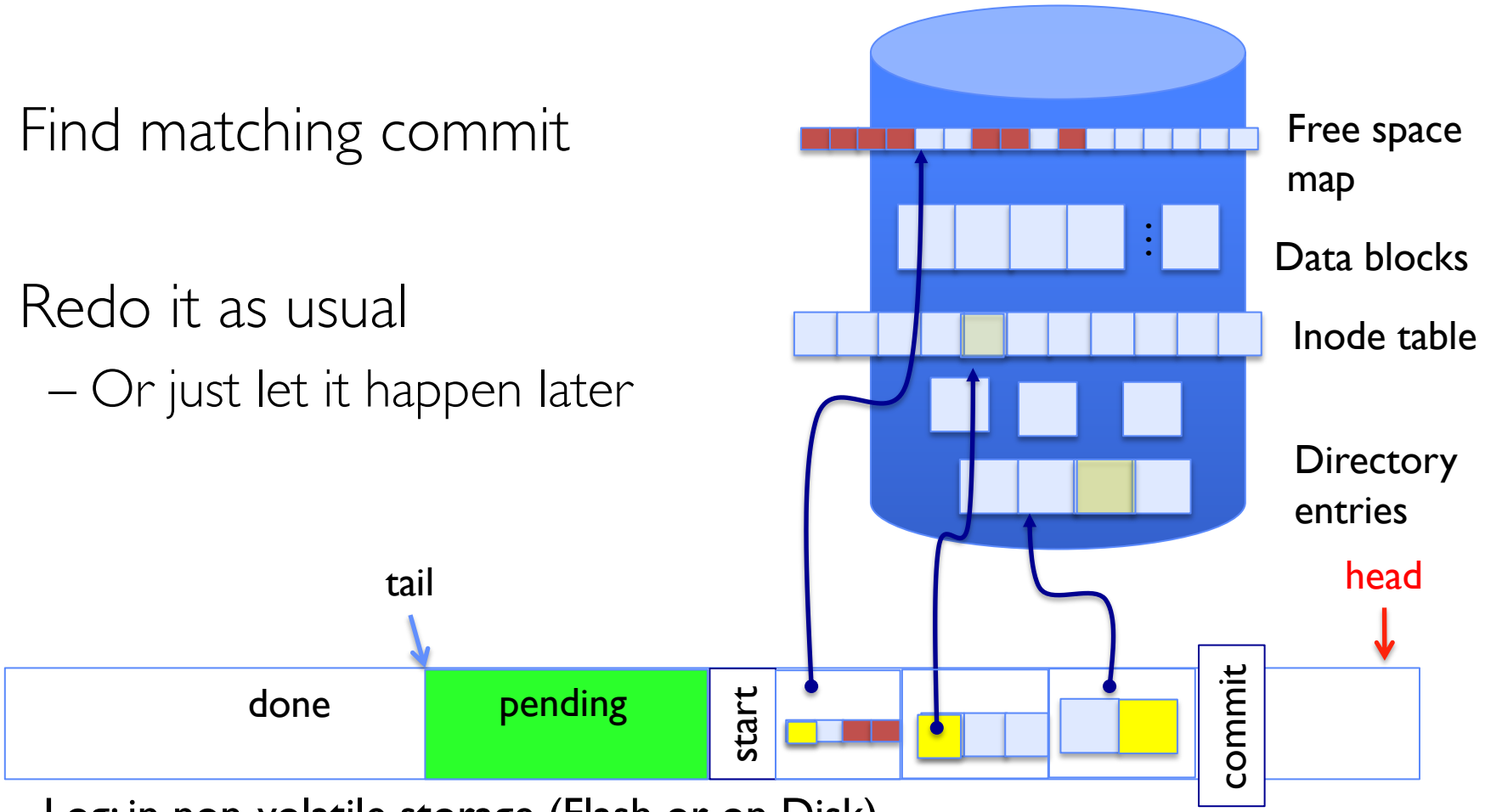
Crash During Logging – Recover

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



Recovery After Commit

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



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 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

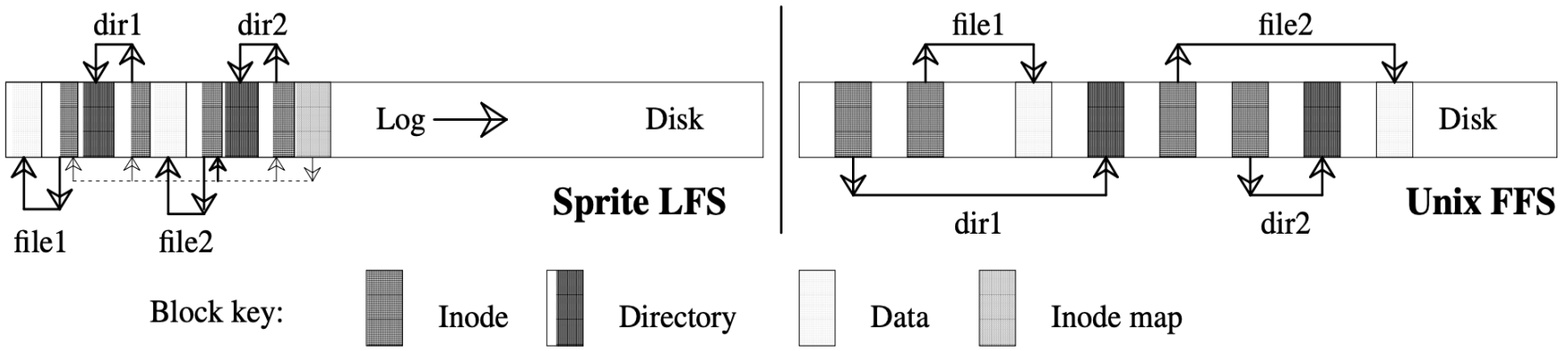


Figure 1 — A comparison between Sprite LFS and Unix FFS.

- 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
- Buffer cache likely to hold information in both cases
  - But disk IOs are very different

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
- Deep interactions between mem management, file system, sharing
  - `mmap()`: map file or anonymous segment to memory
- 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