CS162
Operating Systems and Systems Programming
Lecture 23

Filesystems 3: Buffer Cache, Reliability, Transactions (start)

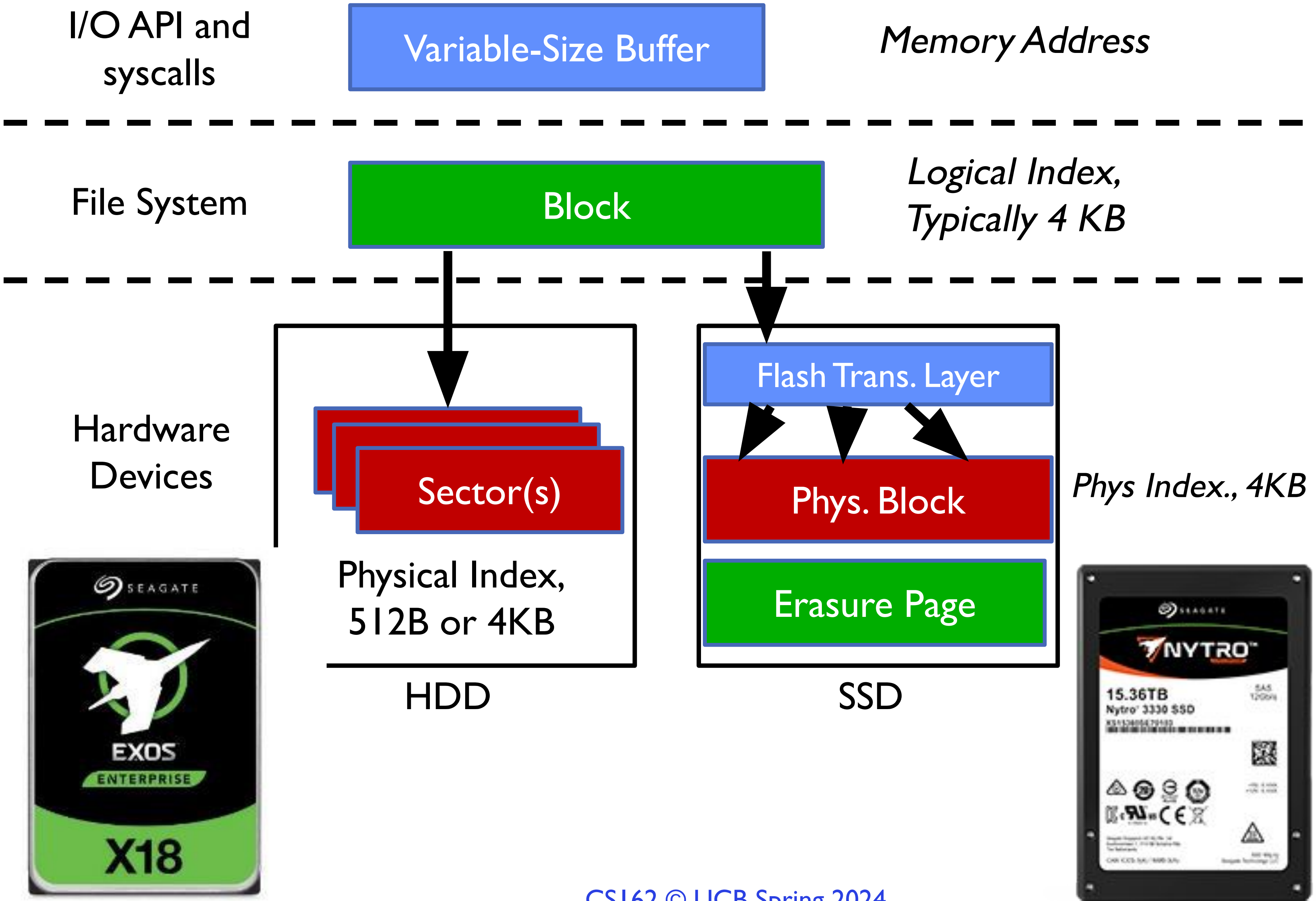
November 21st, 2024

Prof. Ion Stoica

http://cs162.eecs.Berkeley.edu

THE BUFFER CACHE

Recall: From Storage to File Systems

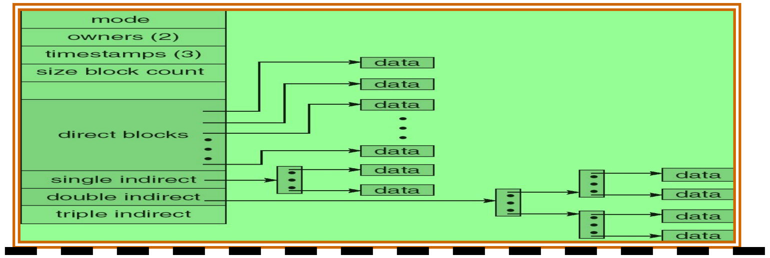


Need for Cache Between FileSystem and Devices

I/O API and syscalls — Variable-Size Buffer — *Memory Address*

Not block-sized or block-aligned access

File System (Block Based)



Logical Index, Typically 4 KB

Reuse of inodes, indirect blocks, data blocks

Buffer Cache (Block Based)

free free

Speed up access to file system path and data

Hardware Devices

Sector(s)

Physical Index, 512B or 4KB

HDD

Flash Trans. Layer

Phys. Block

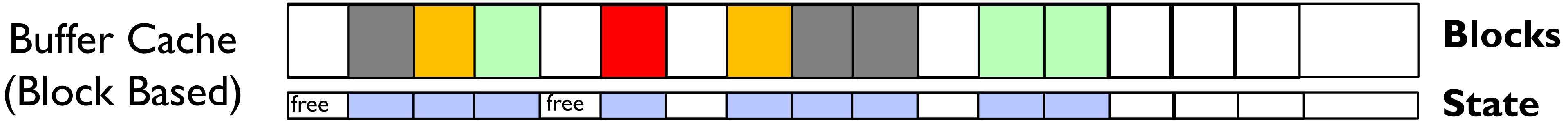
Erasure Page

SSD

Phys Index., 4KB

Apparent speed and flexibility is greater because of Buffer Cache

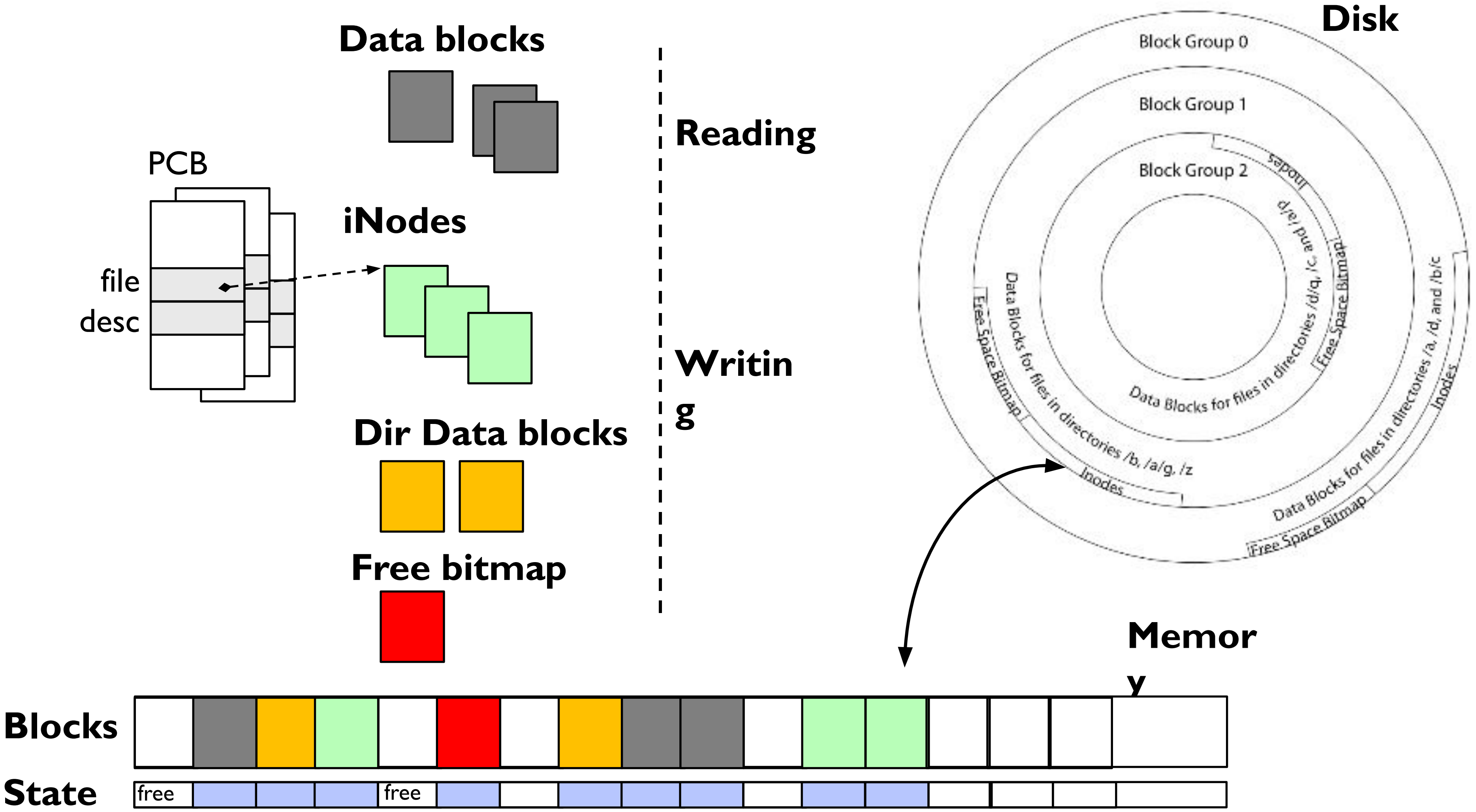
Buffer Cache: Motivation



- Kernel *must* copy disk blocks to memory (somewhere) to access their contents and write them back if modified
  - Could be data blocks, inodes, directory contents, etc.
  - Possibly dirty (modified and not yet 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 (with modifications not on disk)

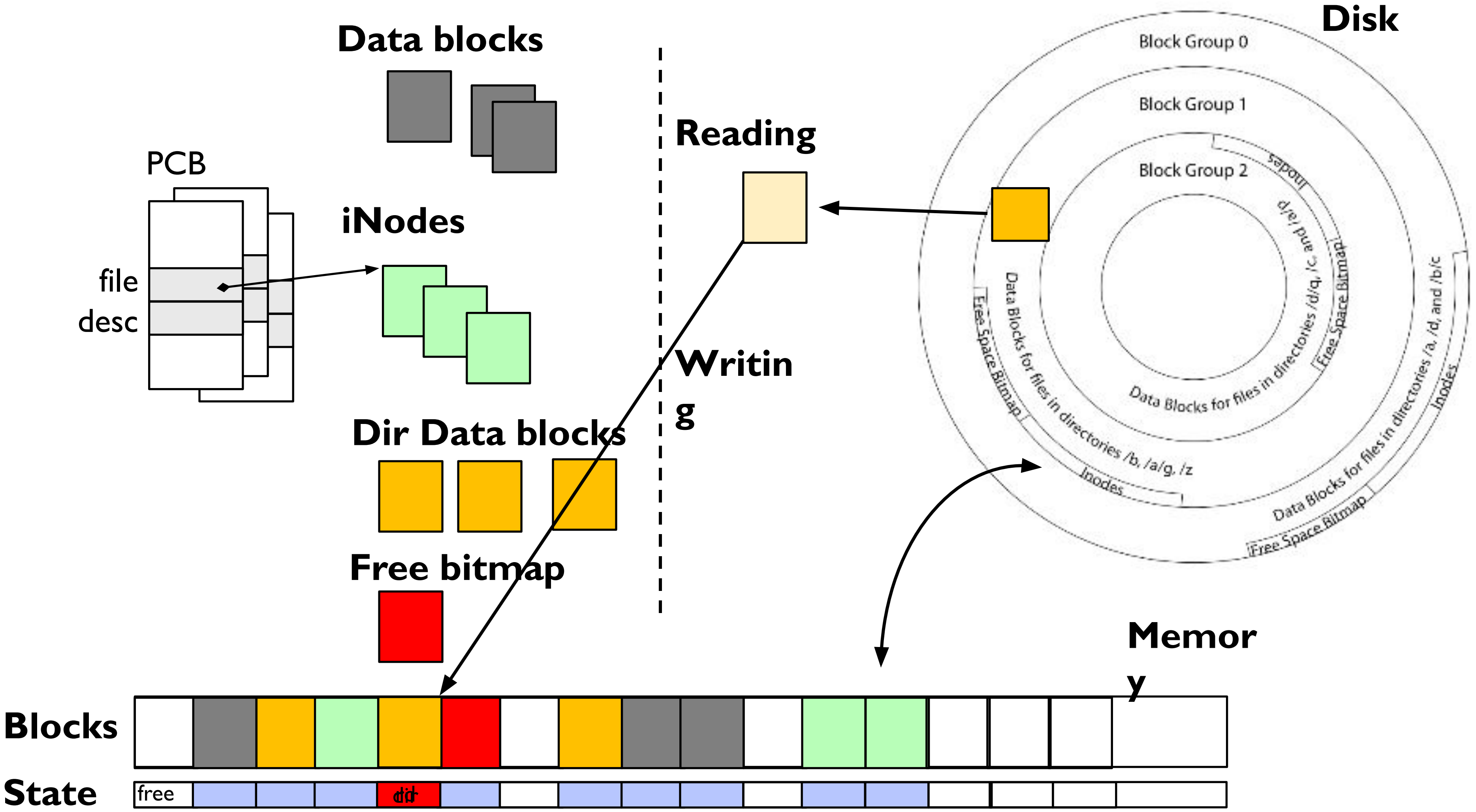
File System Buffer Cache

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



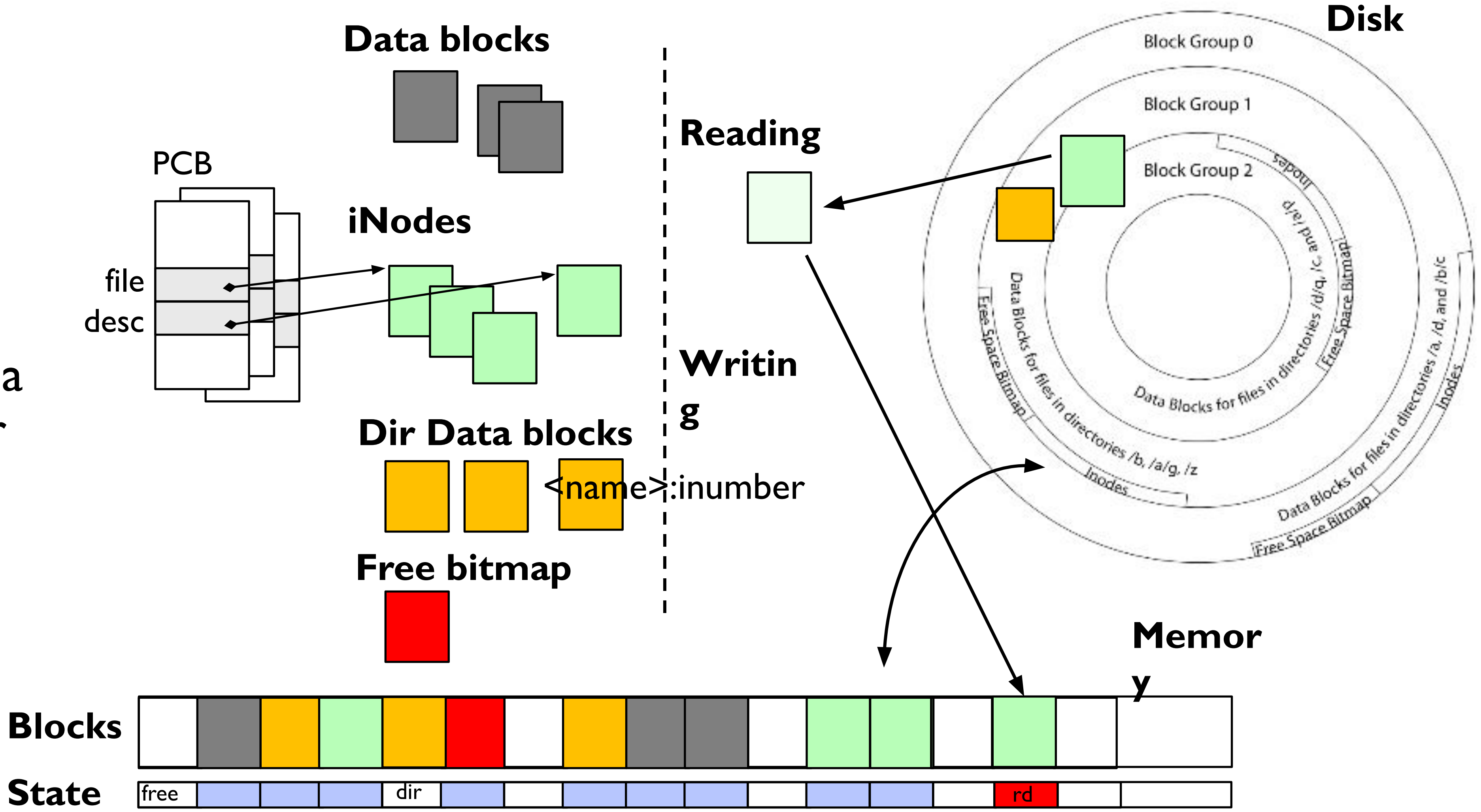
File System Buffer Cache: open

- Directory lookup repeat as needed:
  - load block of directory
  - search for map



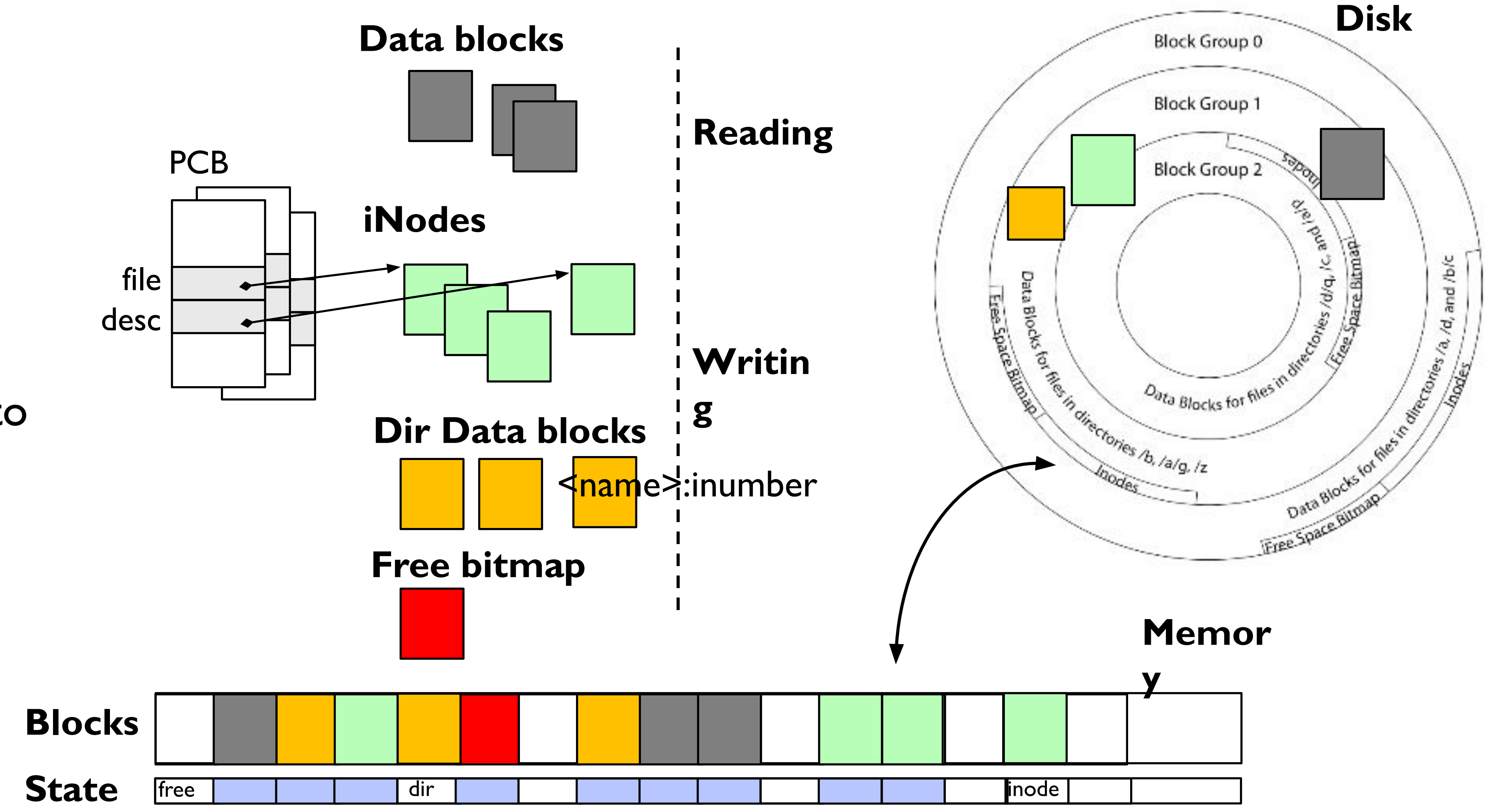
File System Buffer Cache: open

- Directory lookup repeat as needed:
  - load block of directory
  - search for map
- Create reference via open file descriptor



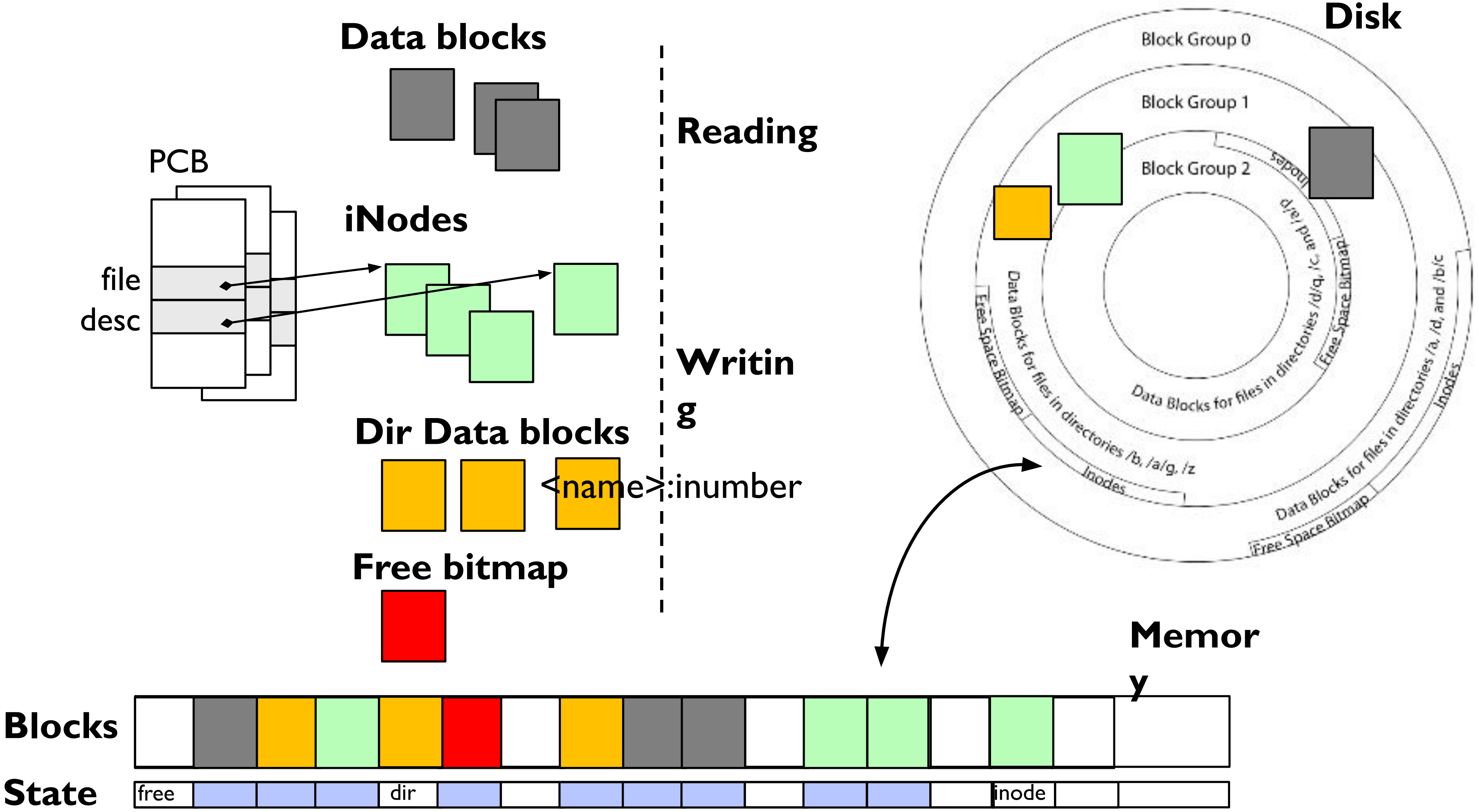
File System Buffer Cache: Read?

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



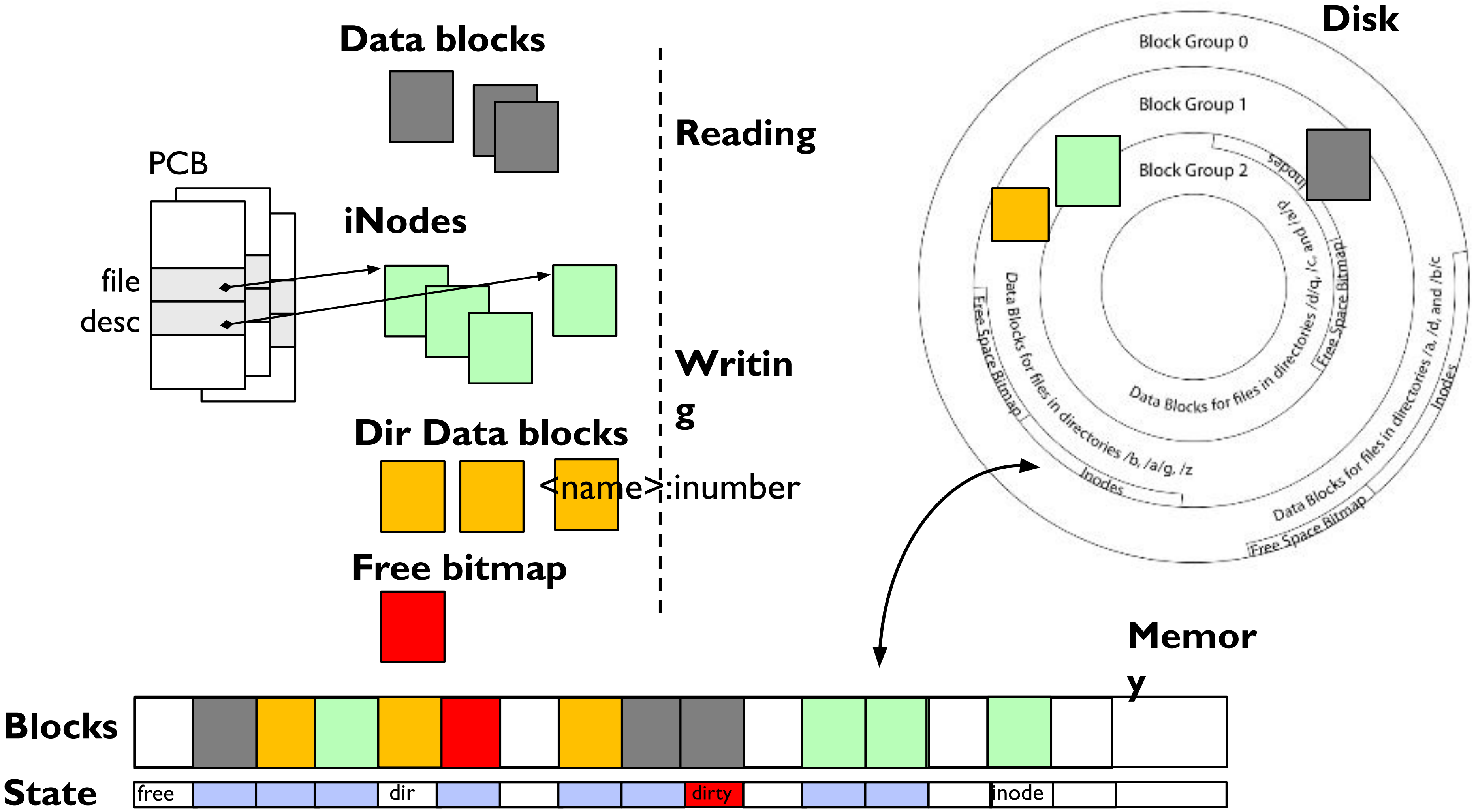
File System Buffer Cache: Write?

- 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
  - 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 Prefetching

- 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
  - Elevator algorithm can efficiently interleave prefetches from concurrent applications
- How much to prefetch?
  - Too much prefetching imposes delays on requests by other applications
  - Too little prefetching causes many seeks (and rotational delays) among concurrent file requests

Delayed Writes

- Buffer cache is a writeback cache (writes are termed "Delayed Writes")

- `write()` copies data from user space to kernel buffer cache
  – Quick return to user space

- `read()` is fulfilled by the cache, so `reads` see the results of `writes`
  – Even if the data has not reached disk

- When does data from a `write` syscall finally reach disk?
  – When the buffer cache is full (e.g., we need to evict something)
  – When the buffer cache is flushed periodically (in case we crash)

Delayed Writes (Advantages)

- Performance advantage: return to user quickly without writing to disk!
- Disk scheduler can efficiently order lots of requests
  - Elevator Algorithm can rearrange writes to avoid random seeks
- 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!

Buffer Caching vs. Demand Paging

- Replacement Policy?
  - Demand Paging: LRU is infeasible; use approximation (like NRU/Clock)
  - Buffer Cache: LRU is OK

- Eviction Policy?
  - Demand Paging: evict not-recently-used pages when memory is close to full
  - Buffer Cache: write back dirty blocks periodically, even if used recently
    - » Why? To minimize data loss in case of a crash

Dealing with Persistent State

- Buffer Cache: write back dirty blocks periodically, even if used recently
  - Why? To minimize data loss in case of a crash
  - Linux does periodic flush every 30 seconds
- Not foolproof! Can still crash with dirty blocks in the cache
  - What if the dirty block was for a directory?
    - » Lose pointer to file's inode (leak space)
    - » **File system now in inconsistent state** ☹️

Takeaway: File systems need recovery mechanisms

HOW TO MAKE FILE SYSTEMS MORE *DURABLE*?

Important "ilities"

- Availability: the probability that the system can accept and process requests
  - Measured in "nines" of probability: e.g. 99.9% probability is "3-nines of availability"
  - Key idea here is independence of failures

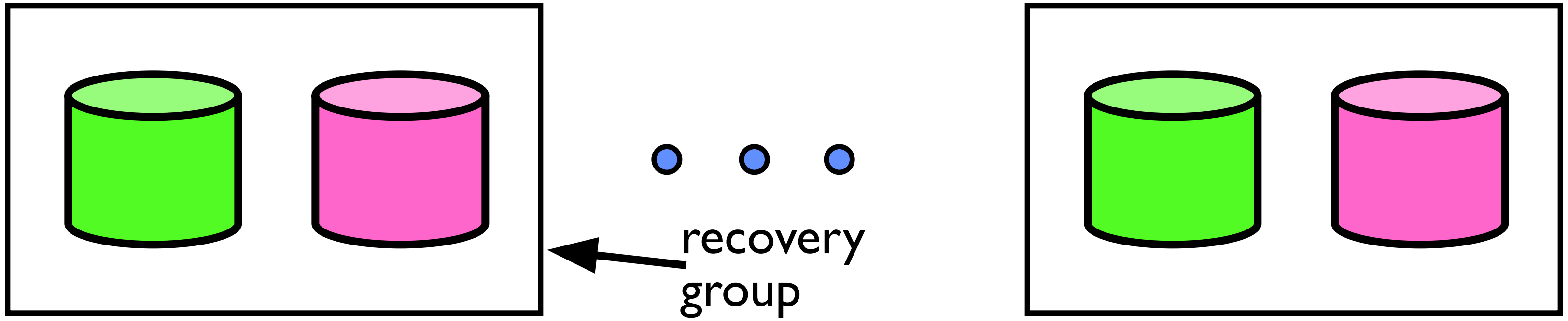
- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone

- 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 Systems more 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

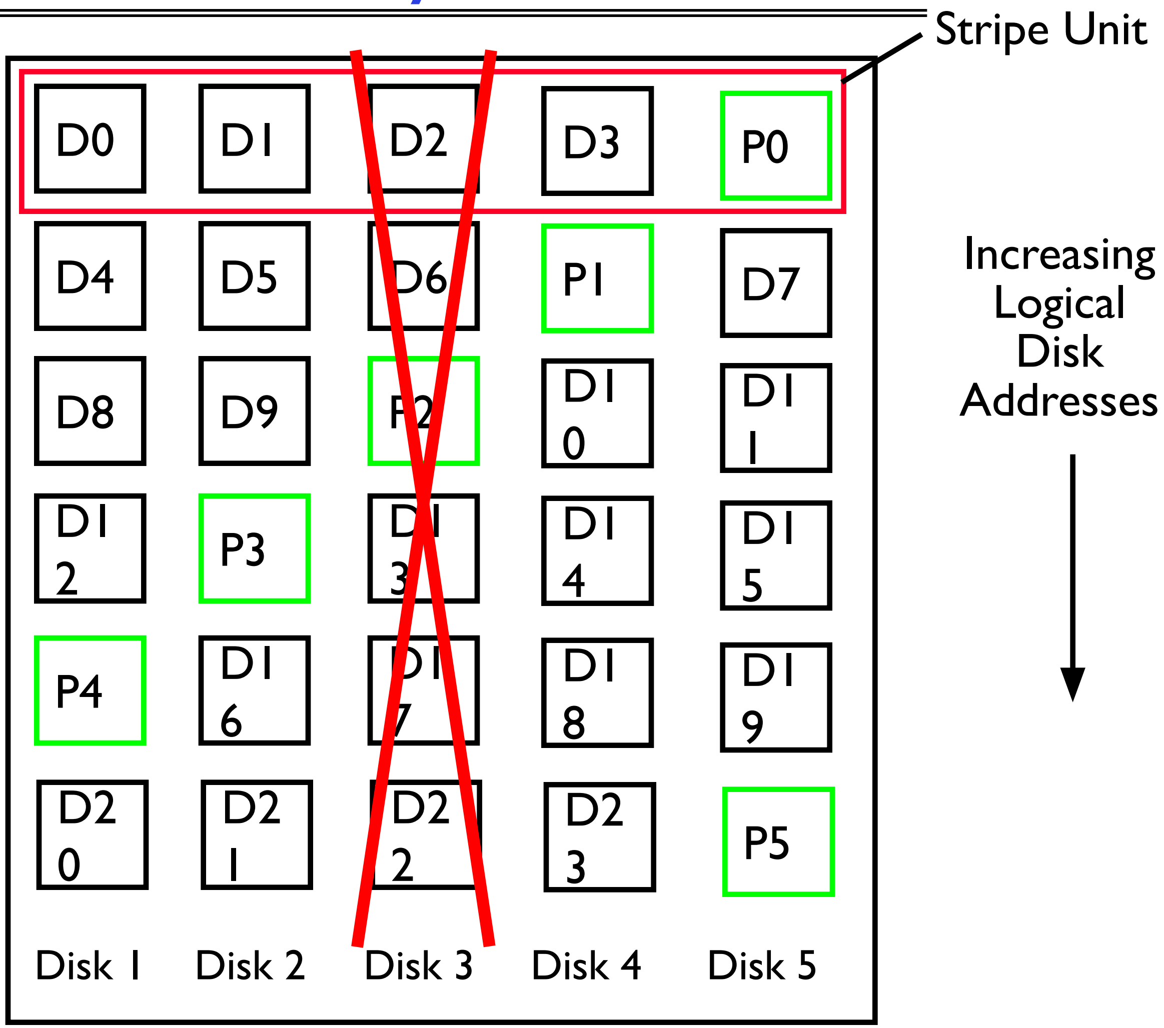


Redundant Array of Inexpensive Disks
(developed here at Berkeley!)

- 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 synchronized (challenging)
- 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 attached to system 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 blocks 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

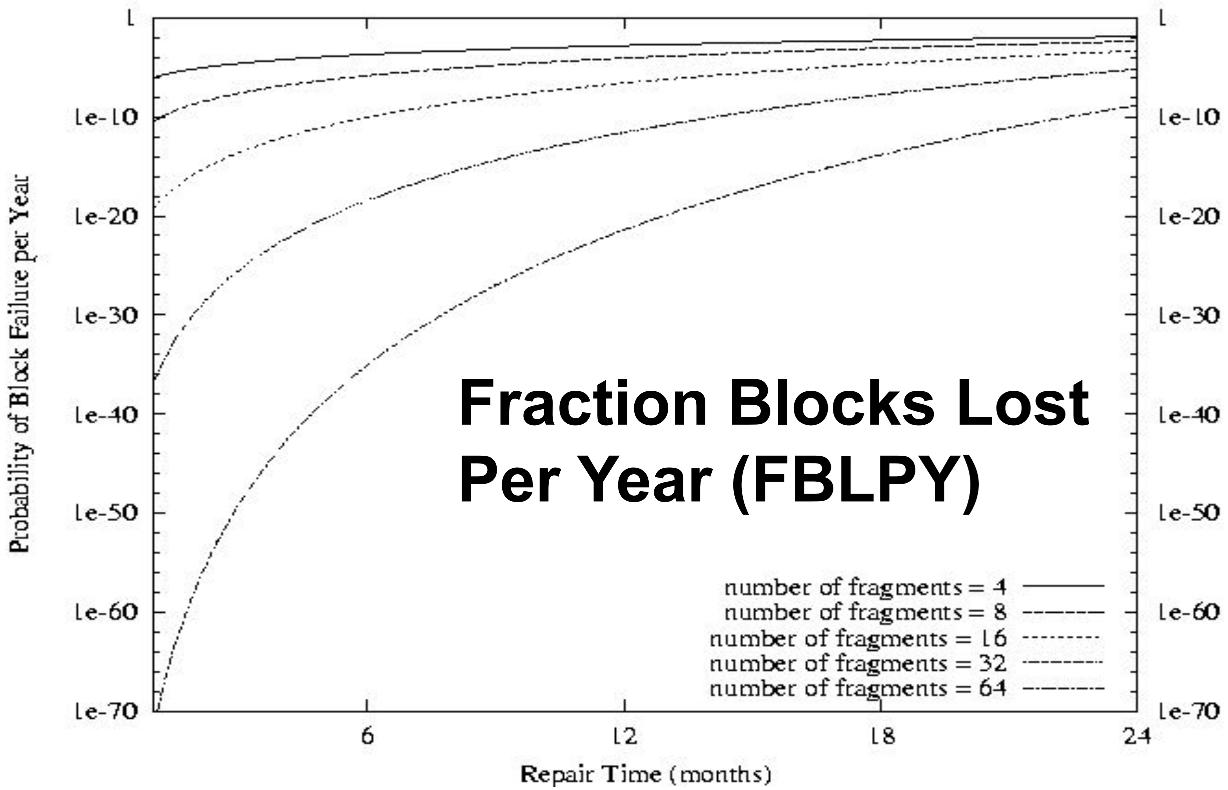


- Can spread information widely across internet for durability
  - RAID algorithms work over geographic scale

RAID 6 and other Erasure Codes

- 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
  - Requires more complex erasure code, such as EVENODD code (see readings)
- More general option for general erasure code: Reed-Solomon codes
  - m data fragements
  - generate n - m extra fragments
  - can tolerate $n - m$ failures
- Erasure codes not just for disk arrays. For example, geographic replication
  - E.g., split data into $m = 4$ fragments, generate $n = 16$ fragments and distribute across Internet
  - Any 4 fragments can be used to recover the original data --- very durable!

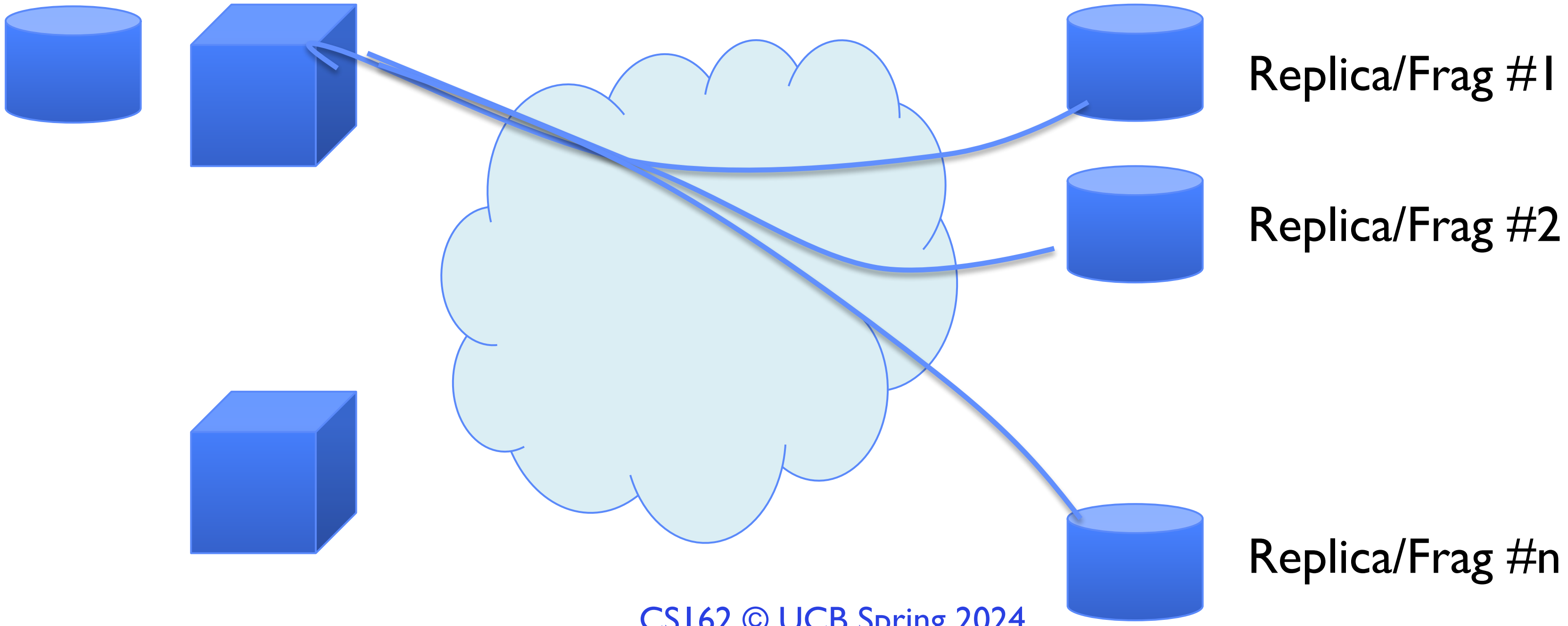
Use of Erasure Coding for High Durability/overhead ratio!



- 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 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

- But durability is not quite enough…!

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
  - Data block ⇐ inode ⇐ free ⇐ directory
- 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)

Berkeley 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

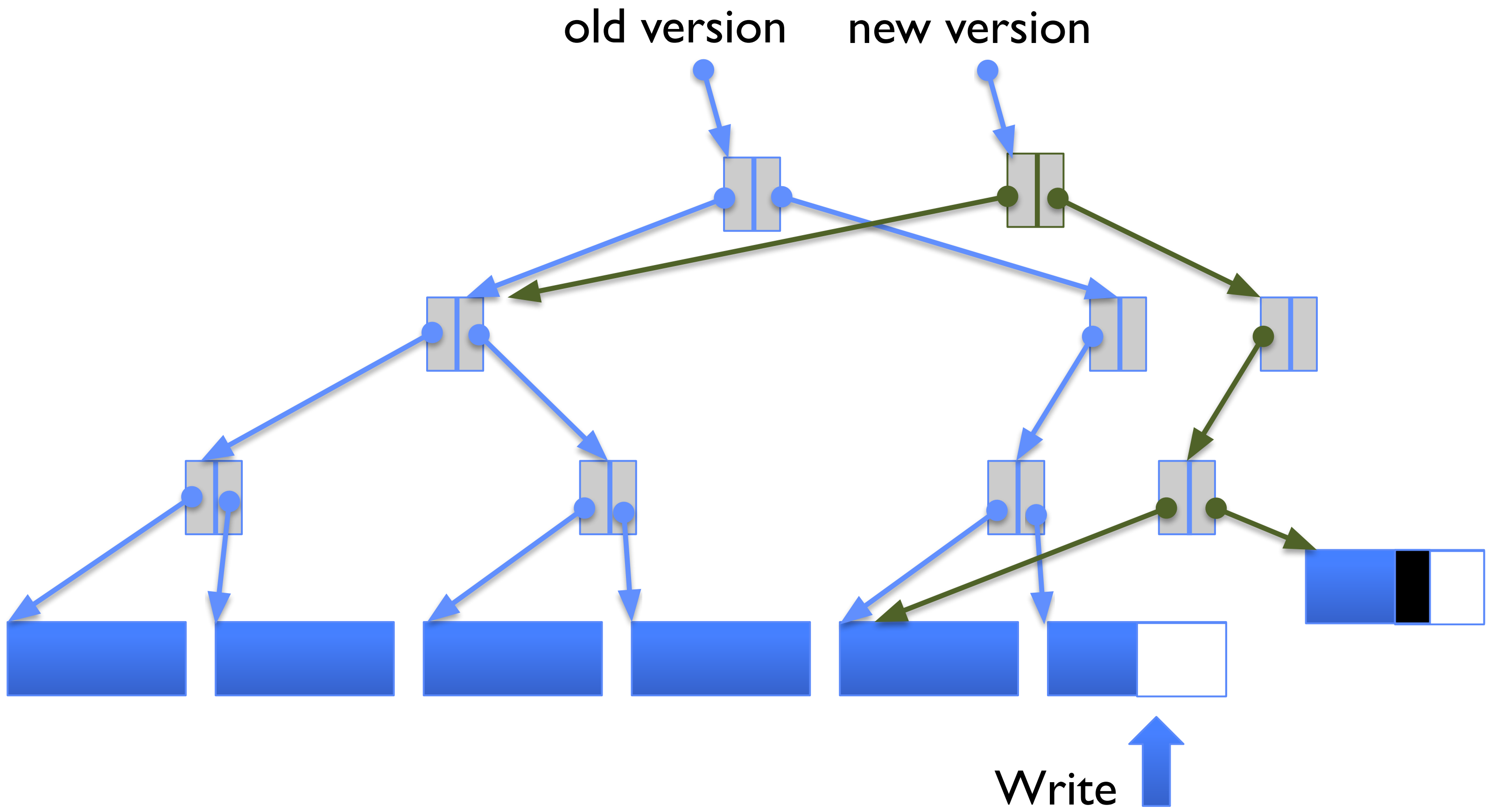
Reliability Approach #2: Copy on Write File Layout

- Recall: multi-level index structure lets us find the data blocks of a file
- Instead of over-writing existing data blocks and updating the index structure:
  - Create a new version of the file with the updated data
  - Reuse blocks that don't change much of what is already in place
  - This is called: Copy On Write (COW)

- 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

Example: 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

Announcements

- Project 3: Design doc due Sunday (11/24)

- Homework 5: Checkpoint deadline Tuesday (11/26)

- Midterm 3: Thursday, 12/05
  - Everything fair game with focus on last 1/3 of class
  - Three *hand-written* cheat-sheets, double sided

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

- A *transaction* is an atomic sequence of reads and writes that takes the system from consistent state to another.

- Recall: Code in a critical section appears atomic to other threads
- Transactions extend the concept of atomic updates from *memory* to *persistent storage*

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

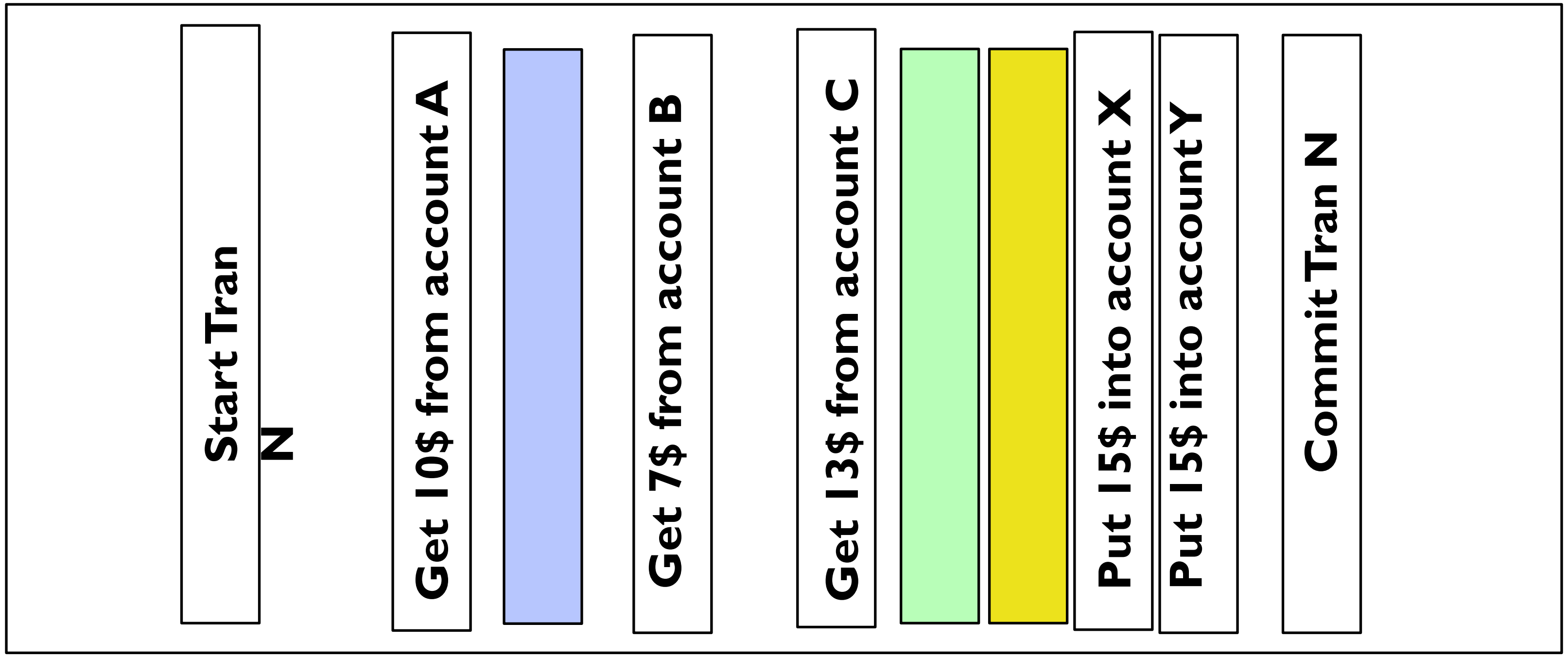
"Classic" Example: Transaction

```
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

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
  - 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 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 to file system
  - e.g. modify inode pointers and directory mapping
- Garbage collection: once a change is applied, remove its entry from the log

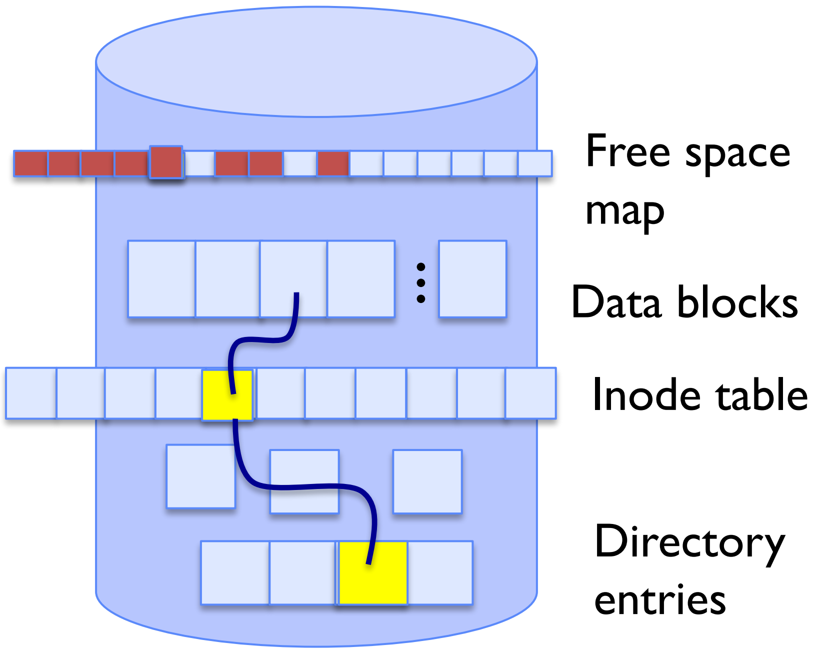
- Linux took original FFS-like file system (ext2) and added a journal to get ext3!
  - Some options: whether or not to write all data to journal or just metadata

- Other examples: NTFS, Apple HFS+/apfs, Linux XFS, JFS, ext4

Creating a File (No Journaling Yet)

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

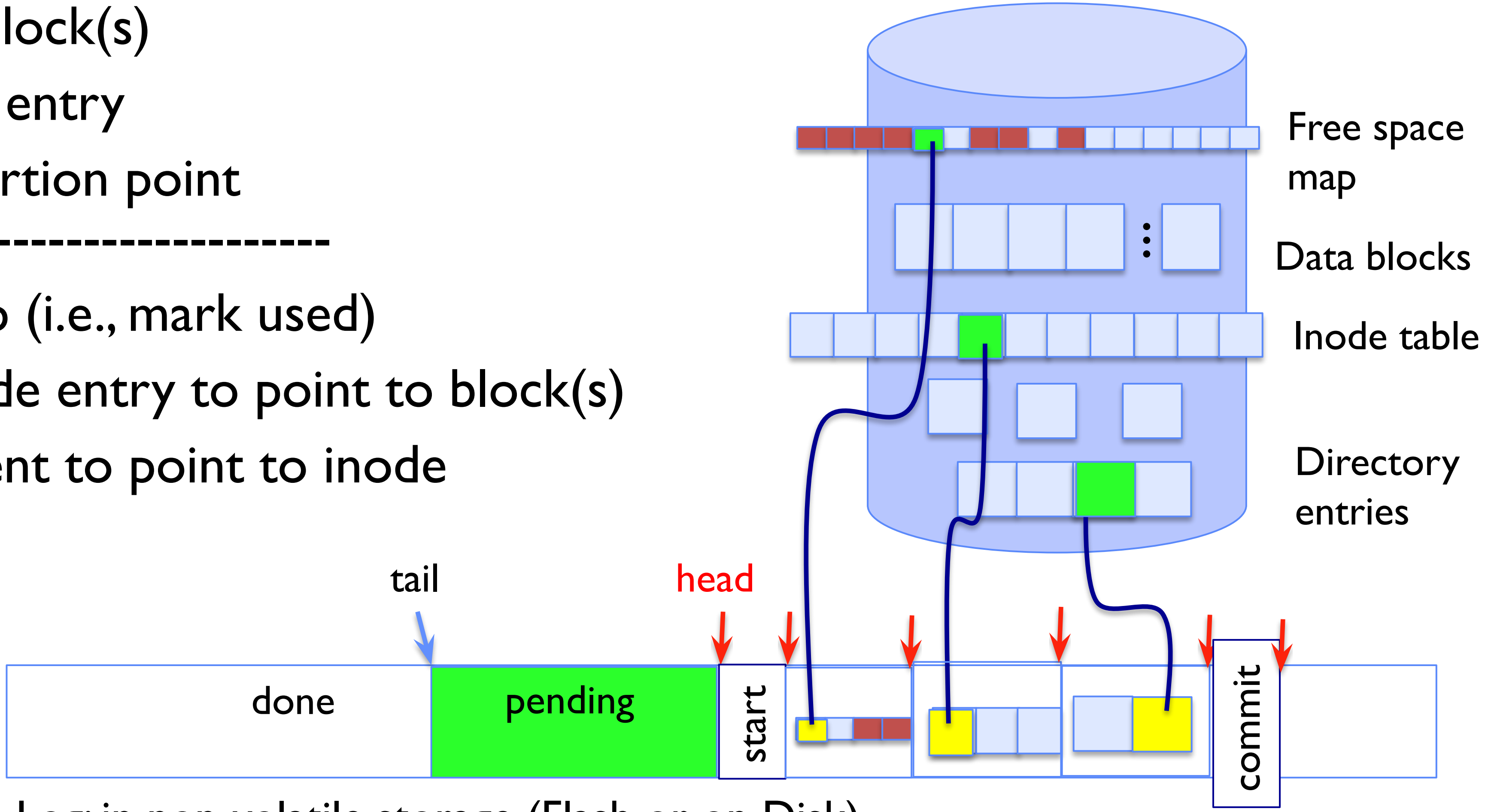
- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



Creating a File (With Journaling)

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

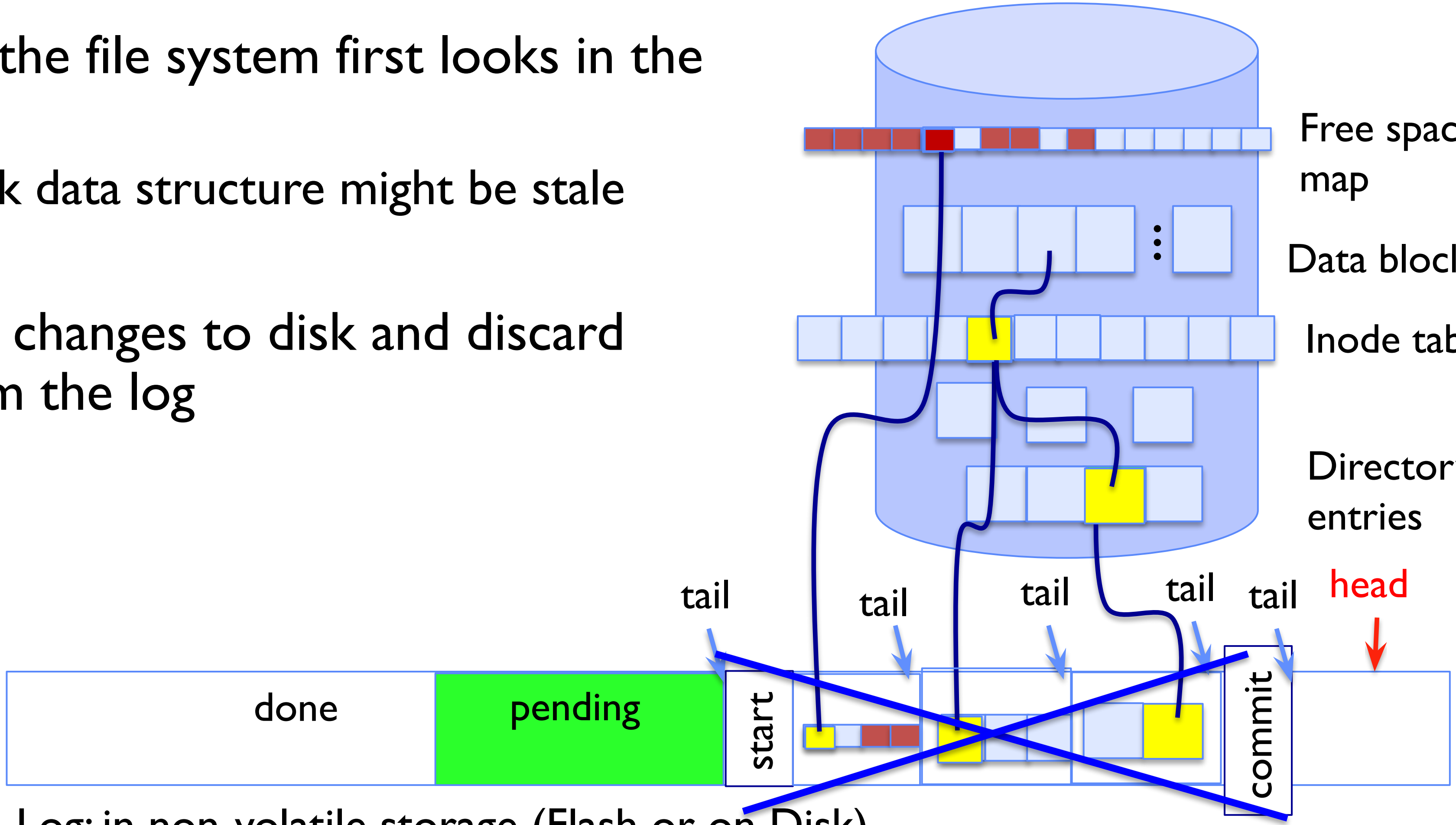
- [log] Write map (i.e., mark used)
- [log] Write inode entry to point to block(s)
- [log] Write dirent to point to inode



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

After Commit, Eventually Replay Transaction

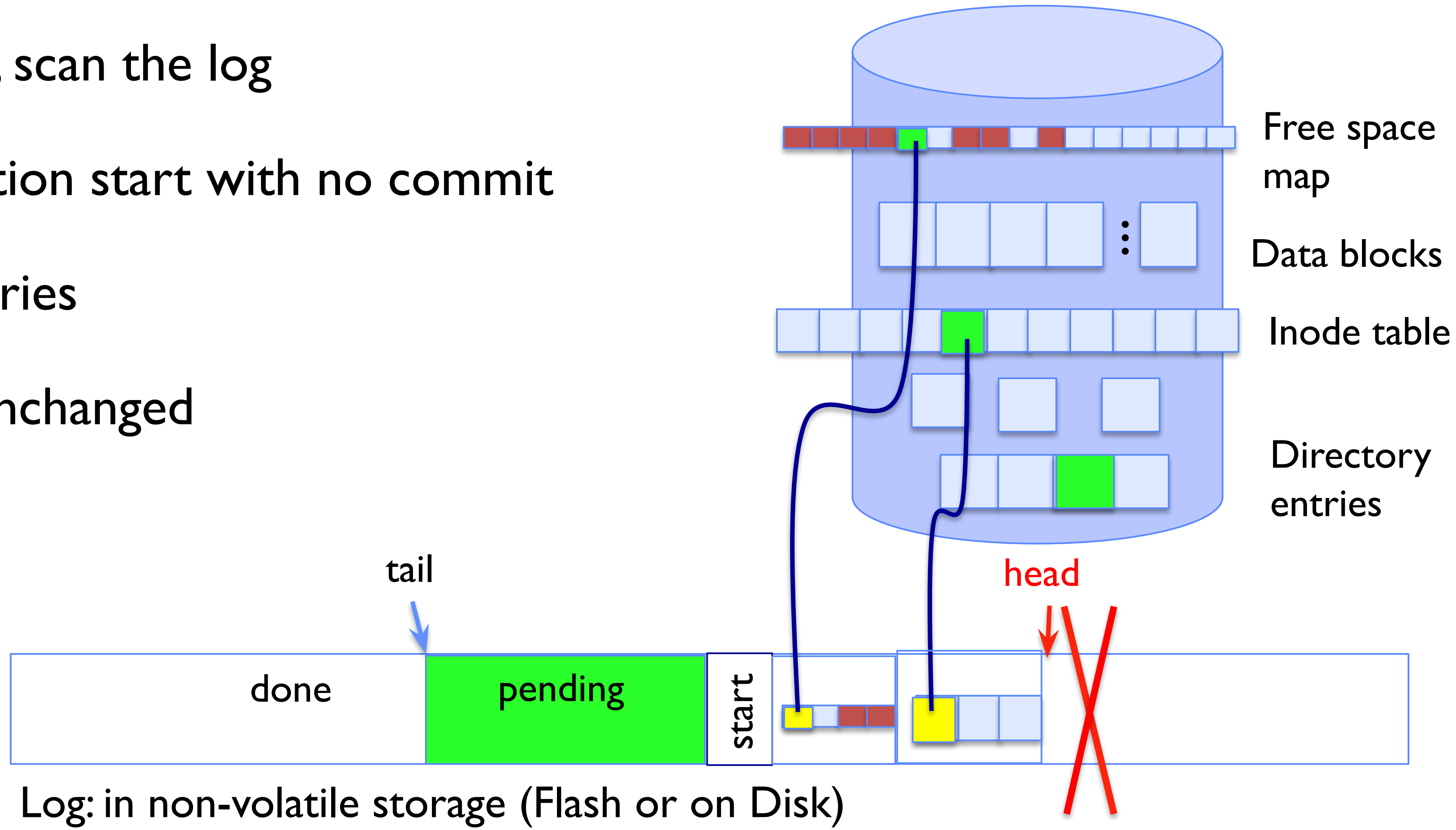
- All accesses to the file system first looks in the log
  - Actual on-disk data structure might be stale
- Eventually, copy changes to disk and discard transaction from the log



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

Crash Recovery: Discard Partial Transactions

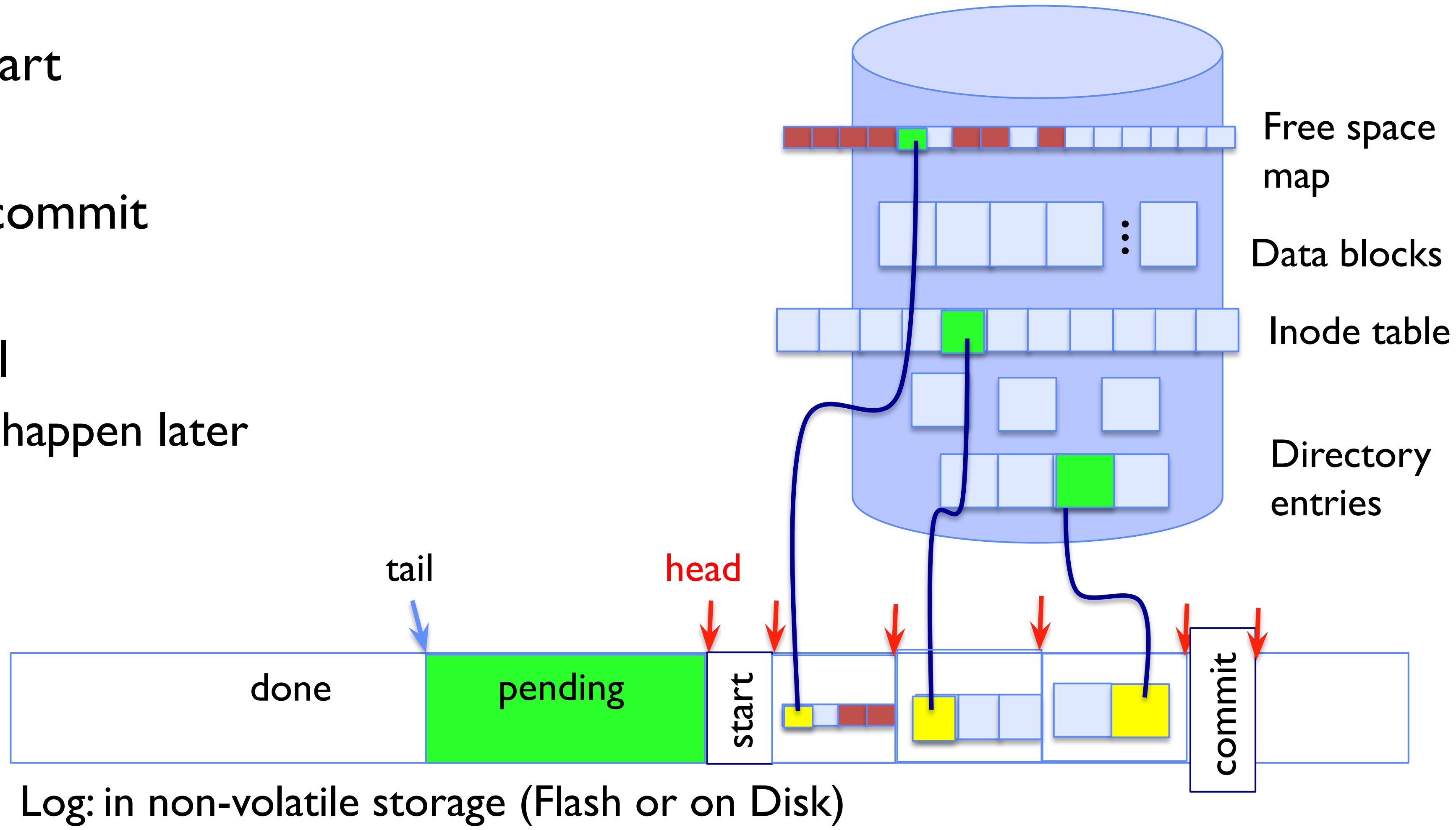
- Upon recovery, scan the log
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



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

Crash Recovery: Keep Complete Transactions

- 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?

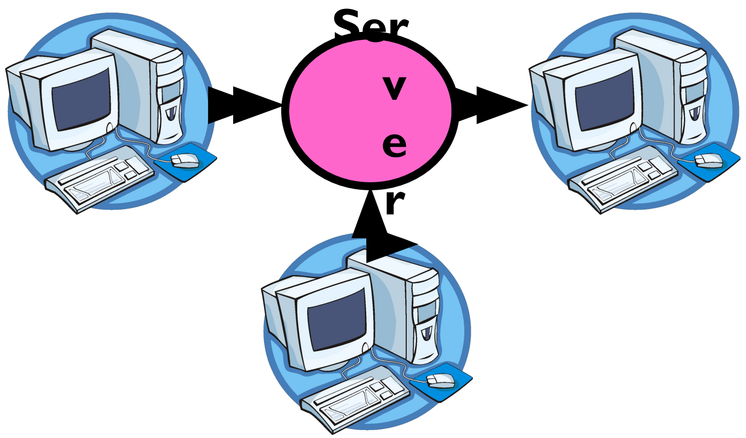
- Updates atomic, even if we crash:
  - Update either gets fully applied or discarded
  - All physical operations *treated as a logical unit*

Isn't this expensive?

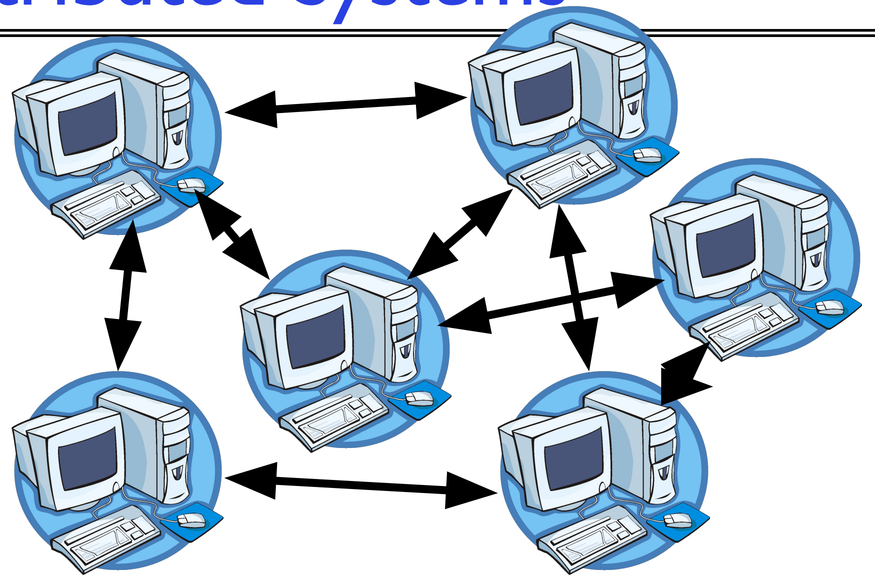
- Yes! We're now writing all data twice (once to log, once to actual data blocks in target file)
- Modern filesystems journal metadata updates only
  - Record modifications to file system data structures
  - But apply updates to a file's contents directly

DISTRIBUTED SYSTEMS

Centralized vs Distributed Systems



Client/Server Model



Peer-to-Peer Model

- Centralized System: major functions performed by a single physical computer
  – Originally, everything on single computer
  – Later: client/server model
- Distributed System: physically separate computers working together on task
  – Early model: multiple servers working together
    » Probably in the same room or building
    » Often called a "cluster"
  – Later models: peer-to-peer/wide-spread collaboration

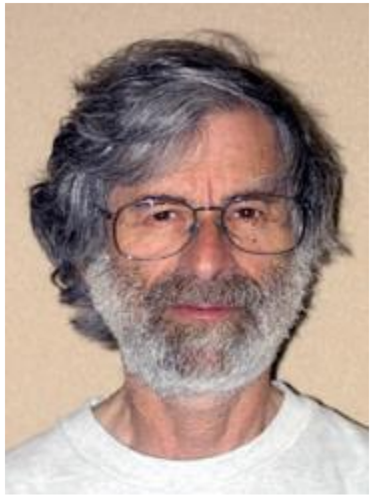
Distributed Systems: Motivation/Issues/Promise

- Why do we want distributed systems?
  - Cheaper and easier to build lots of simple computers
  - More resources (such as cluster of GPUs for training)
  - Easier to add resources incrementally
  - Users can have complete control over some components
  - Collaboration: much easier for users to collaborate through network resources (such as network file systems)

- The *promise* of distributed systems:
  - *Higher availability*: one machine goes down, use another
  - *Better durability*: store data in multiple locations
  - *More security*: each piece easier to make secure

Distributed Systems: Reality

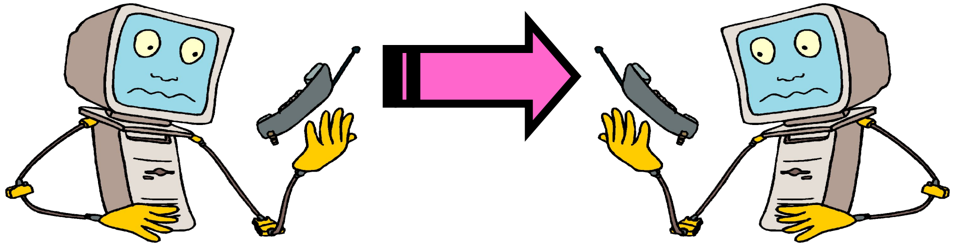
- Reality has been disappointing
  - *Worse availability*: depend on every machine being up
    - » Lamport: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."
  - *Worse reliability*: can lose data if any machine crashes
  - *Worse security*: anyone in world can break into system
- Coordination is more difficult
  - Must coordinate multiple copies of shared state information
  - What would be easy in a centralized system becomes a lot more difficult
- Trust/Security/Privacy/Denial of Service
  - Many new variants of problems arise as a result of distribution
  - Can you trust the other members of a distributed application enough to even perform a protocol correctly?
  - Corollary of Lamport's quote: "A distributed system is one where you can't do work because some computer you didn't even know existed is successfully coordinating an attack on my system!"



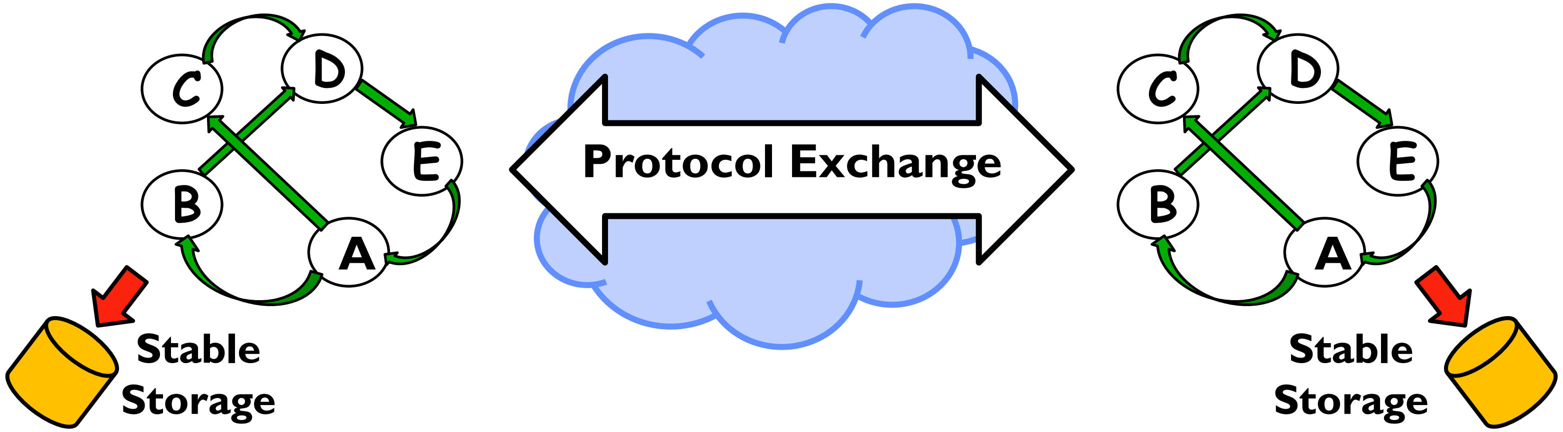
Leslie Lamport

Distributed Systems: Goals/Requirements

- Transparency: the ability of the system to mask its complexity behind a simple interface
- Possible transparencies:
  - Location: Can't tell where resources are located
  - Migration: Resources may move without the user knowing
  - Replication: Can't tell how many copies of resource exist
  - Concurrency: Can't tell how many users there are
  - Parallelism: System may speed up large jobs by splitting them into smaller pieces
  - Fault Tolerance: System may hide various things that go wrong
- Transparency and collaboration require some way for different processors to communicate with one another



How do entities communicate? A Protocol!



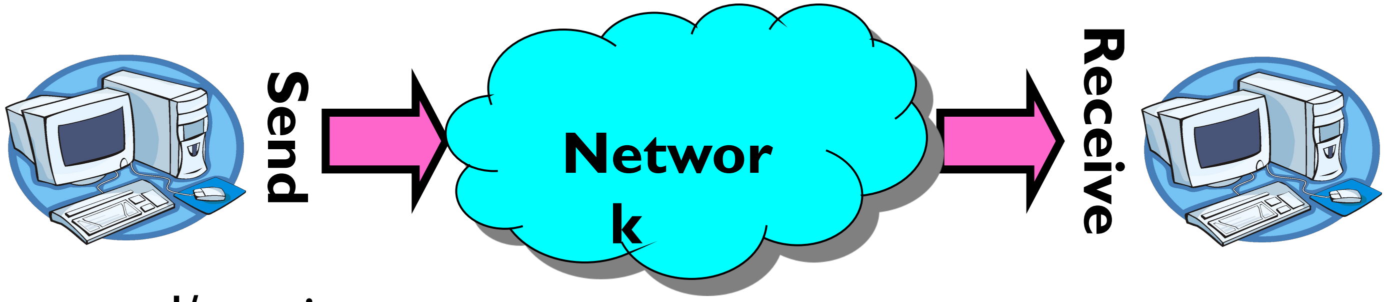
- A protocol is an agreement on how to communicate, including:
  - Syntax: how a communication is specified & structured
    - » Format, order messages are sent and received
  - Semantics: what a communication means
    - » Actions taken when transmitting, receiving, or when a timer expires
- Described formally by a state machine
  - Often represented as a message transaction diagram
  - Can be a partitioned state machine: two parties synchronizing duplicate sub-state machines between them
  - Stability in the face of failures!

Examples of Protocols in Human Interactions

- Telephone
  1. (Pick up / open up the phone)
  2. Listen for a dial tone / see that you have service
  3. Dial
  4. Should hear ringing …
  5. Callee: "Hello?"
  6. Caller: "Hi, it's Anthony…."
     Or: "Hi, it's me" (← what's *that* about?)
  7. Caller: "Hey, do you think … blah blah blah …" **pause**
  8. Callee: "Yeah, blah blah blah …" **pause**
  9. Caller: Bye
  10. Callee: Bye
  11. Hang up

Distributed Applications

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines
    - » No shared memory, so cannot use test&set



  - One Abstraction: send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (mbox): temporary holding area for messages
    - » Includes both destination location and queue
    - » Over Internet, destination specified by IP address and Port (Recall Web server example!)
  - Send(message,mbox)
    - » Send message to remote mailbox identified by mbox
  - Receive(buffer,mbox)
    - » Wait until mbox has message, copy into buffer, and return
    - » If threads sleeping on this mbox, wake up one of them

Using Messages: Send/Receive behavior

- When should `send(message,mbox)` return?
  - When receiver gets message? (i.e. ack received)
  - When message is safely buffered on destination?
  - Right away, if message is buffered on source node?
- Actually two questions here:
  - When can the sender be sure that receiver actually received the message?
  - When can sender reuse the memory containing message?
- Mailbox provides 1-way communication from T1→T2
  - T1→buffer→T2
  - Very similar to producer/consumer
    - » Send = V, Receive = P
    - » However, can't tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

```
Producer:
    int msg1[1000];
    while(1) {
        prepare message;
        send(msg1,mbox);
    }
```

Send Message

```
Consumer:
    int buffer[1000];
    while(1) {
        receive(buffer,mbox);
        process message;
    }
```

Receive Message

- No need for producer/consumer to keep track of space in mailbox: handled by send/receive
  – This is one of the roles of the window in TCP: window is size of buffer on far end
  – Restricts sender to forward only what will fit in buffer

Messaging for Request/Response communication

- What about two-way communication?
  - Request/Response
    - » Read a file stored on a remote machine
    - » Request a web page from a remote web server
  - Also called: client-server
    - » Client ≡ requester, Server ≡ responder
    - » Server provides "service" (file storage) to the client
- Example: File service

```
Client: (requesting the file)
   char response[1000];

   send("read rutabaga", server_mbox);
   receive(response, client_mbox);

Server: (responding with the file)
   char command[1000], answer[1000];

   receive(command, server_mbox);
   decode command;
   read file into answer;
   send(answer, client_mbox);
```

Request File

Get Response

Receive Request

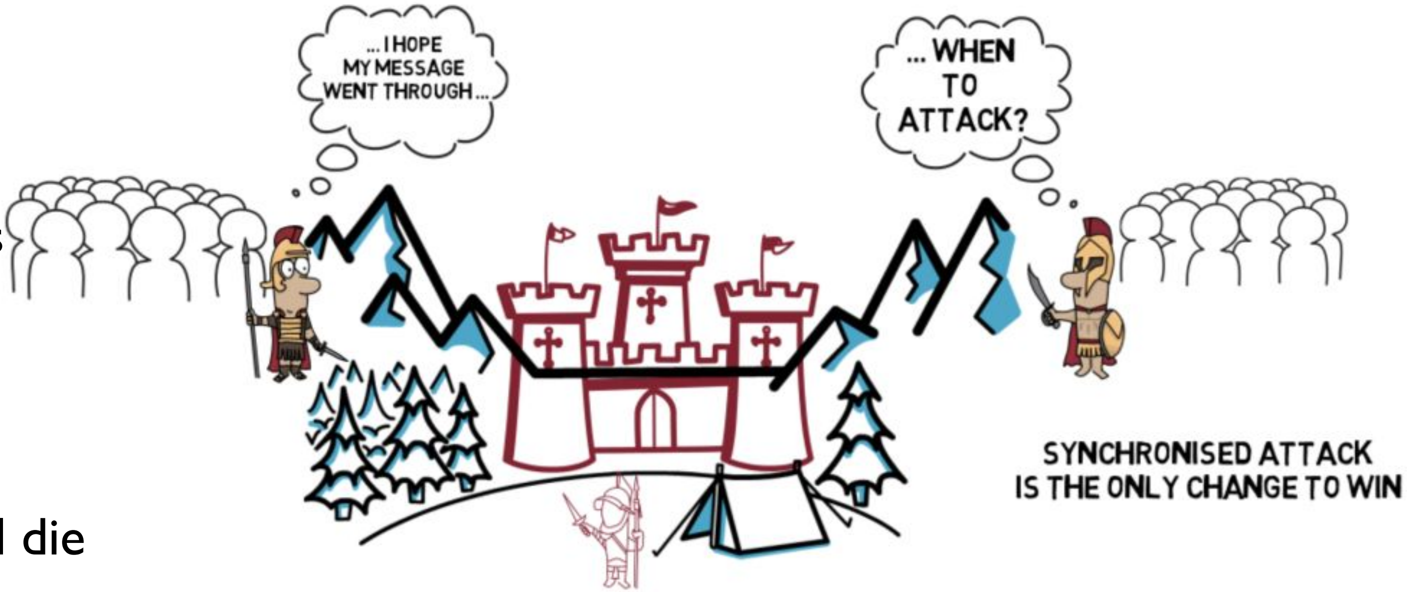
Send Response

Distributed Consensus Making

- Consensus problem
  - All nodes propose a value
  - Some nodes might crash and stop responding
  - Eventually, all remaining nodes decide on the same value from set of proposed values
- Distributed Decision Making
  - Choose between "true" and "false"
  - Or Choose between "commit" and "abort"
- Equally important (but often forgotten!): make it durable!
  - How do we make sure that decisions cannot be forgotten?
    - » This is the "D" of "ACID" in a regular database
  - In a global-scale system?
    - » What about erasure coding or massive replication?
    - » Like BlockChain applications!

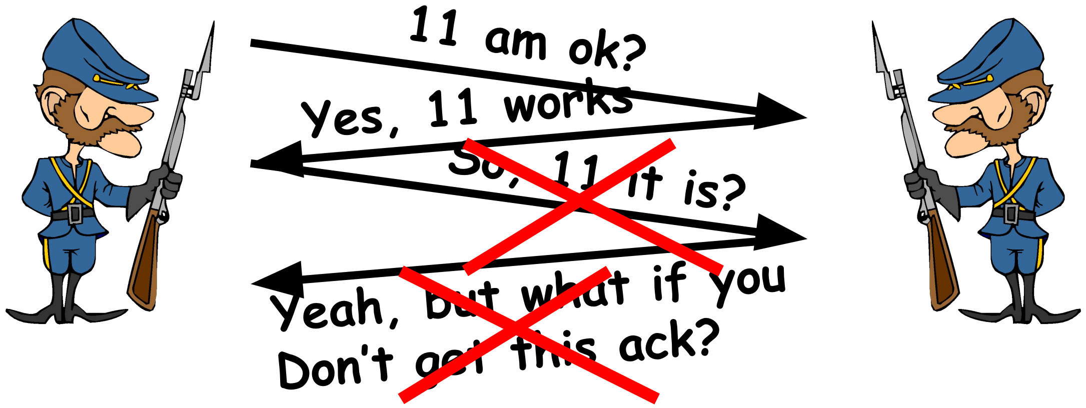
General's Paradox

- General's paradox:
  - Constraints of problem:
    - » Two generals, on separate mountains
    - » Can only communicate via messengers
    - » Messengers can be captured
  - Problem: need to coordinate attack
    - » If they attack at different times, they all die
    - » If they attack at same time, they win
  - Named after Custer, who died at Little Big Horn because he arrived a couple of days too early



General's Paradox (con't)

- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, "no", even if all messages get through



  - No way to be sure last message gets through!
  - In real life, use radio for simultaneous (out of band) communication
- So, clearly, we need something other than simultaneity!

File System Summary

- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  – Can contain "dirty" blocks (blocks yet on disk)
- RAID
  – User replication (mirroring) and parity bits to protect against disk failures
- 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
  – Journaled file systems such as ext3, NTFS
  – Commit sequence to durable log, then update the disk
  – Log takes precedence over disk
  – Replay committed transactions, discard partials