Operating Systems
Lecture 15

Reliable fs

Prof. Mengwei Xu

Threats to FS Reliability

- Operation interruption
  - A crash or power failure
  - A file operation often consists of many I/O updates to the storage
  - An example: *mv ./dir1/file1 ./dir2/file2*

Threats to FS Reliability

- Operation interruption
  - A crash or power failure
  - A file operation often consists of many I/O updates to the storage
  - An example: *mv ./dir1/file1 ./dir2/file2*
    - ❑ Writing the dir1 directory file to remove file1
    - ❑ (optional) Growing the dir2 directory's file to include another block of storage to accommodate a new directory entry for file2
    - ❑ Writing the new directory entry to the directory file
    - ❑ Updating the last-modified time of the dir1 directory
    - ❑ Updating the file system's free space bitmap
    - ❑ Updating the size and last-modified time of the dir2 directory
  - At physical level, operations complete one at a time

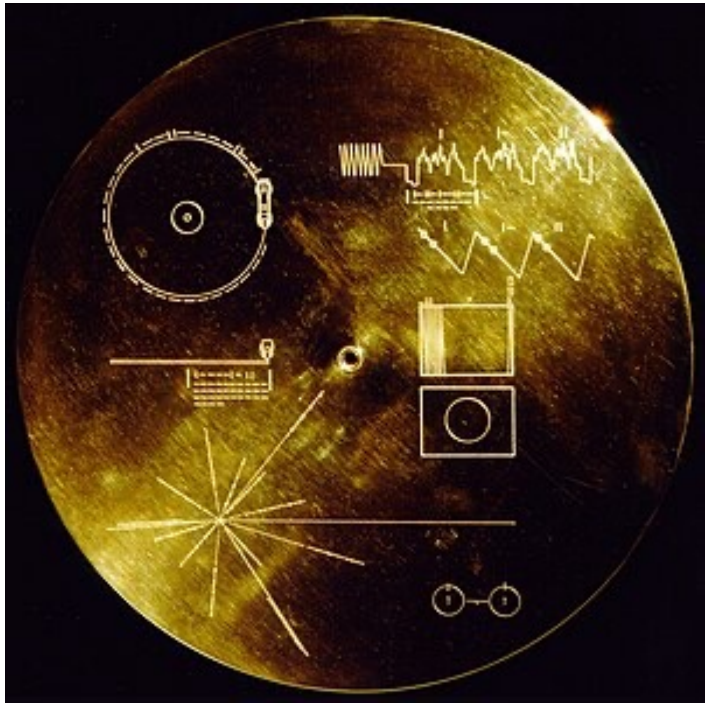
Threats to FS Reliability

- Operation interruption
  - A crash or power failure
  - A file operation often consists of many I/O updates to the storage
  - An example: *mv ./dir1/file1 ./dir2/file2*
  - At physical level, operations complete one at a time

- Loss of stored data
  - Either physical or electric

Reliability vs. Availability

- Reliability (可靠性): the probability that the storage system will continue to be reliable for some specified period of time
- Availability (可用性): the probability that the storage system will be available at any given time

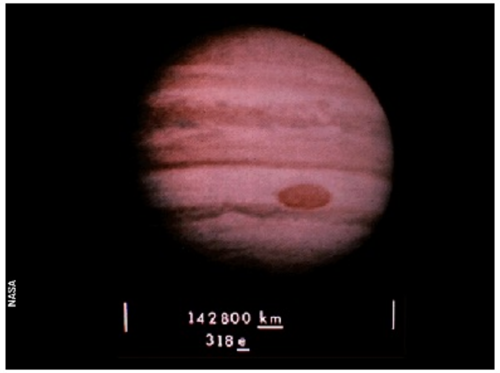


Voyager Golden Record

> This is a present from a small, distant world, a token of our sounds, our science, our images, our music, our thoughts and our feelings. We are attempting to survive our time so we may live into yours.
>
> — Jimmy Carter

Select images on the Voyager Golden Record

A woman in a store



A photo of Jupiter with its diameter indicated

This image depicts humans licking, eating, and drinking as modes of feeding.

What a Reliable FS Does?

- "All or nothing"
  - Either an update is completed, or not at all
  - Must be guaranteed whenever a crash happens
  - Must be transparent to users/apps
  - An example: transfer $100 from Bob's account to Alice's account

- Quite similar to the critical section problem in concurrency
  - Avoid someone observing the state in an intermediate, inconsistent state
  - No control over "when it happens"

Goals for Today

- Transactions for atomic updates
  - Redo Logging
- Redundancy for media failures
  - RAID

Goals for Today

- Transactions for atomic updates
  - Redo Logging
- Redundancy for media failures
  - RAID

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 (file system check, *fsck*) :

- 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

Issues with Approach #1

- Complex reasoning
  - So many possible operations and failures

- Slow updates
  - File systems are forced to insert sync operations or barriers between dependent operations

- Extremely slow recovery
  - Need to scan all of its disks for inconsistent metadata structures

Transactions

- Use *Transactions* (事务) 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

- They extend concept of atomic update from memory to stable storage
  - Atomically update multiple persistent data structures

Transactions

- An atomic sequence of actions (reads/writes) on a storage system (or database)

- That takes it from one consistent state to another

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 Key Properties of Transactions

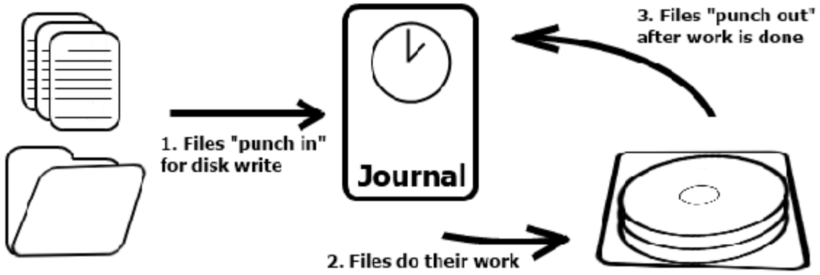
- **Atomicity:** all actions in the transaction happen, or none happen
- **Consistency:** transactions maintain data integrity, e.g.,
  - Balance cannot be negative
  - Cannot reschedule meeting on February 30
- **Isolation:** execution of one transaction is isolated from that of all others; no problems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

Logging

- 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 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
- Basic assumption:
  - Updates to sectors are atomic and ordered

Logging

- Log: an append-only file containing log records
  - <start t>
    - ❑transaction t has begun
  - <t,x,v>
    - ❑transaction t has updated block x and its new value is v
      - Can log block "diffs" instead of full blocks
      - Can log *operations* instead of data
  - <commit t>
    - ❑transaction t has committed – updates will survive a crash



- Committing involves writing the records – the home data needn't be updated at this time
- Logs are often kept in a separation partition
- Once transactions are committed, logs can be cleaned up!

Implementing Transactions: Redo Logging

- Prepare
  - Write all changes/updates to log (日志)
  - Can happen at once, or over time
  - Wait until all updates are written in log
- Commit
  - Append a commit record to the log
  - Or can roll back (abandoned), write a roll-back record
- Write-back
  - Write all of the transaction's updates to disk
- Garbage collection
  - Reclaim space in log
- Recovery
  - Read log
  - Redo any operations for committed transactions
  - Garbage collect log

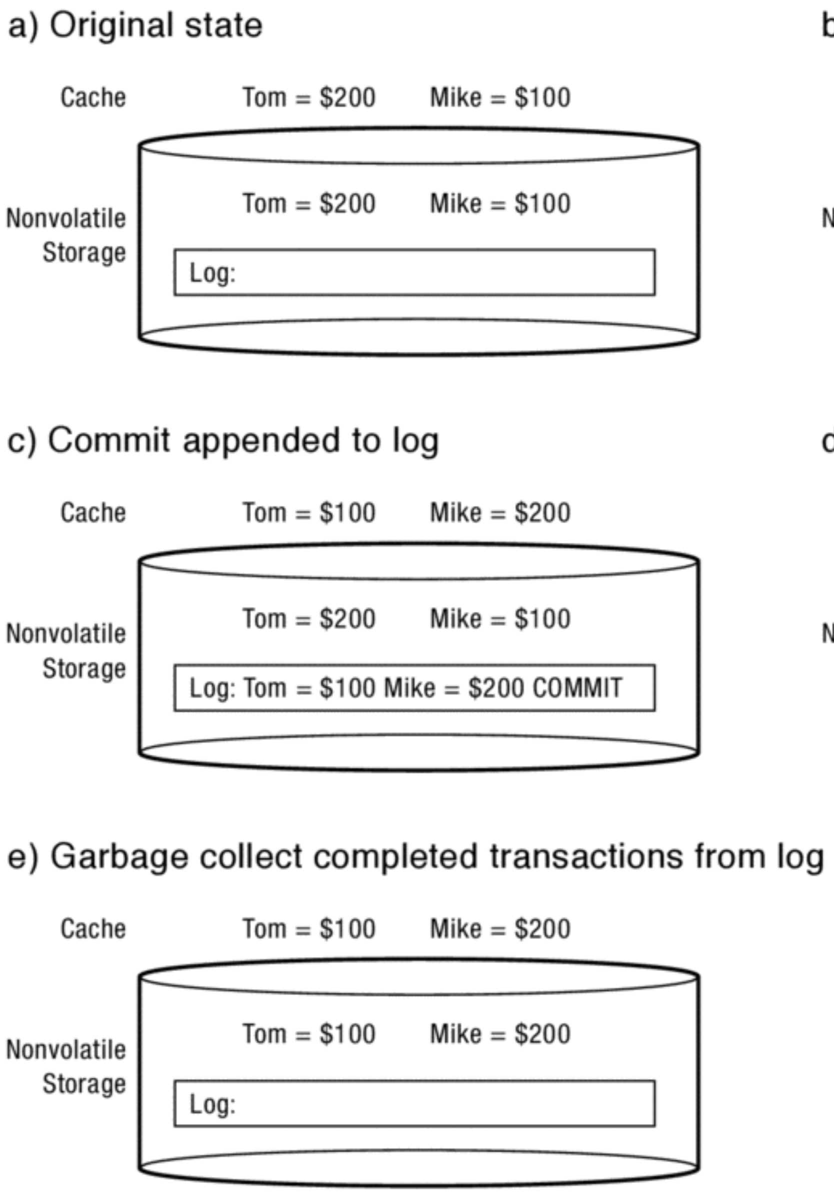
Implementing Transactions: Redo Logging

- Prepare
  - Write all changes/updates to log (日志)
  - Can happen at once, or over time
  - Wait until all updates are written in log
- Commit
  - Append a commit record to the log
  - Or can roll back (abandoned), write a roll-back record
- Write-back
  - Write all of the transaction's updates to disk
- Garbage collection
  - Reclaim space in log

An atomic operation

- Before it, we can safely roll-back
- After it, the transaction must take effect

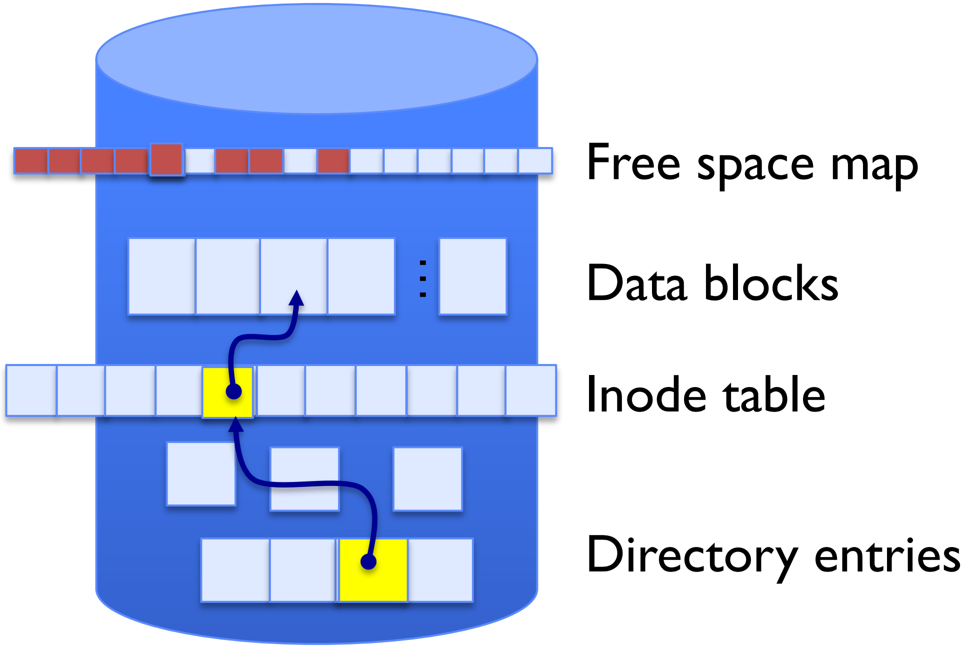
Example #1


a) Original state
Cache
Tom = $200 Mike = $100
Nonvolatile Storage
Tom = $200 Mike = $100
Log:
b) Updates appended to log
Cache
Tom = $100 Mike = $200
Nonvolatile Storage
Tom = $200 Mike = $100
Log: Tom = $100 Mike = $200
c) Commit appended to log
Cache
Tom = $100 Mike = $200
Nonvolatile Storage
Tom = $200 Mike = $100
Log: Tom = $100 Mike = $200 COMMIT
d) Updates applied
Cache
Tom = $100 Mike = $200
Nonvolatile Storage
Tom = $100 Mike = $200
Log: Tom = $100 Mike = $200 COMMIT
e) Garbage collect completed transactions from log
Cache
Tom = $100 Mike = $200
Nonvolatile Storage
Tom = $100 Mike = $200
Log:

Example #2: Creating a File

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

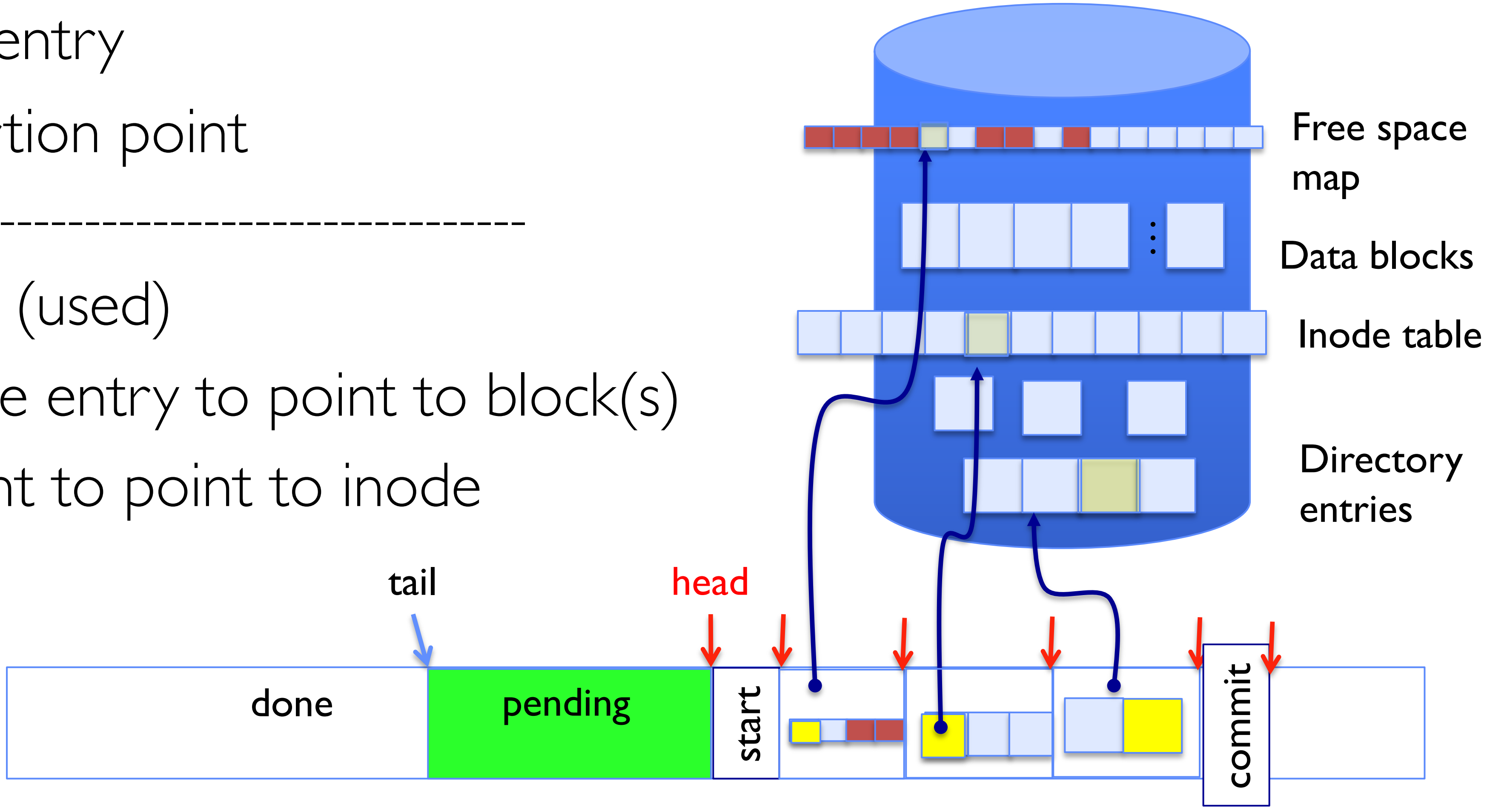
- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



Example #2: Creating a File

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

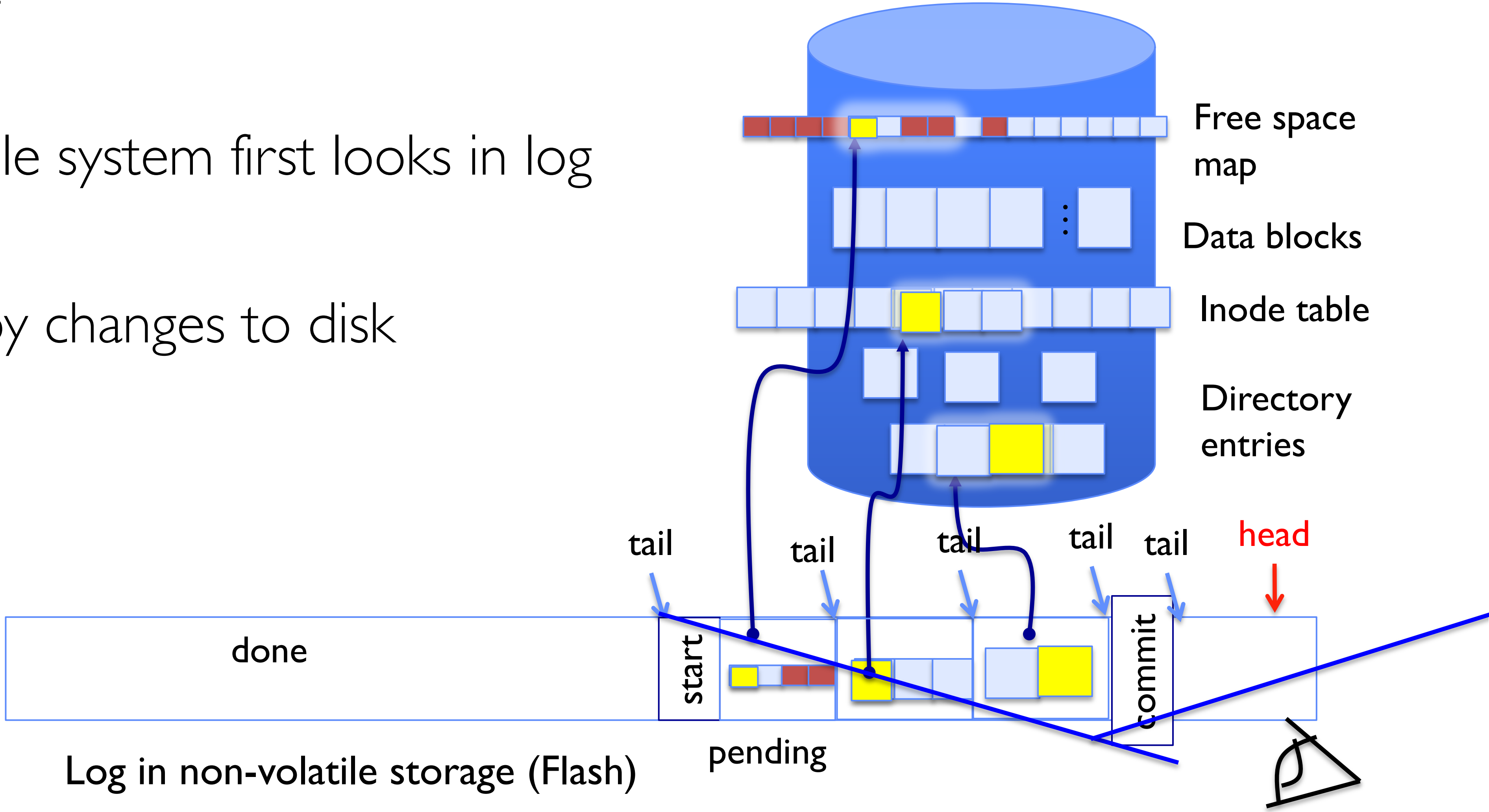
- [log] Write map (used)
- [log] Write inode entry to point to block(s)
- [log] Write dirent to point to inode



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

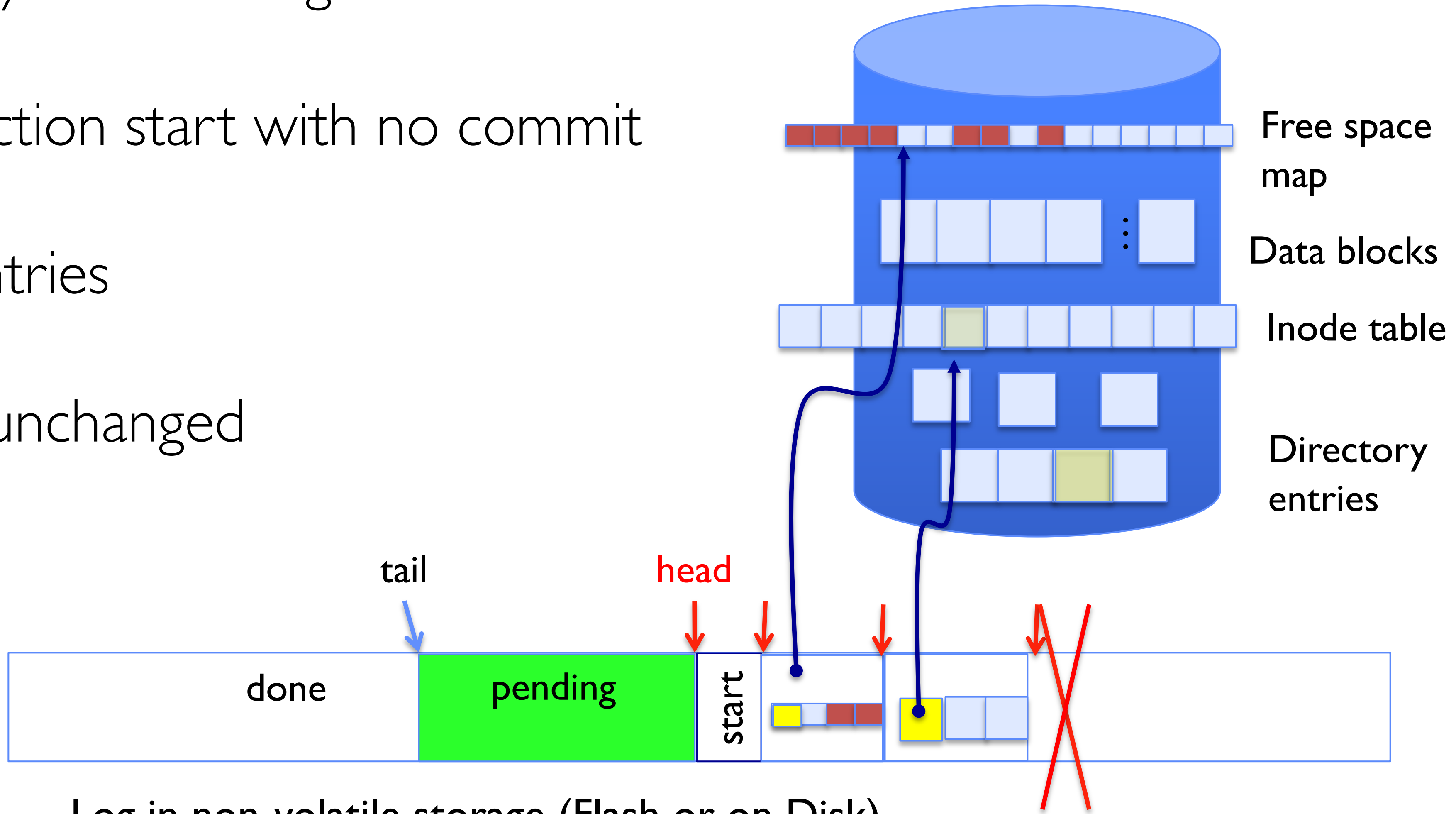
ReDo Log

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



Crash During Logging – Recover

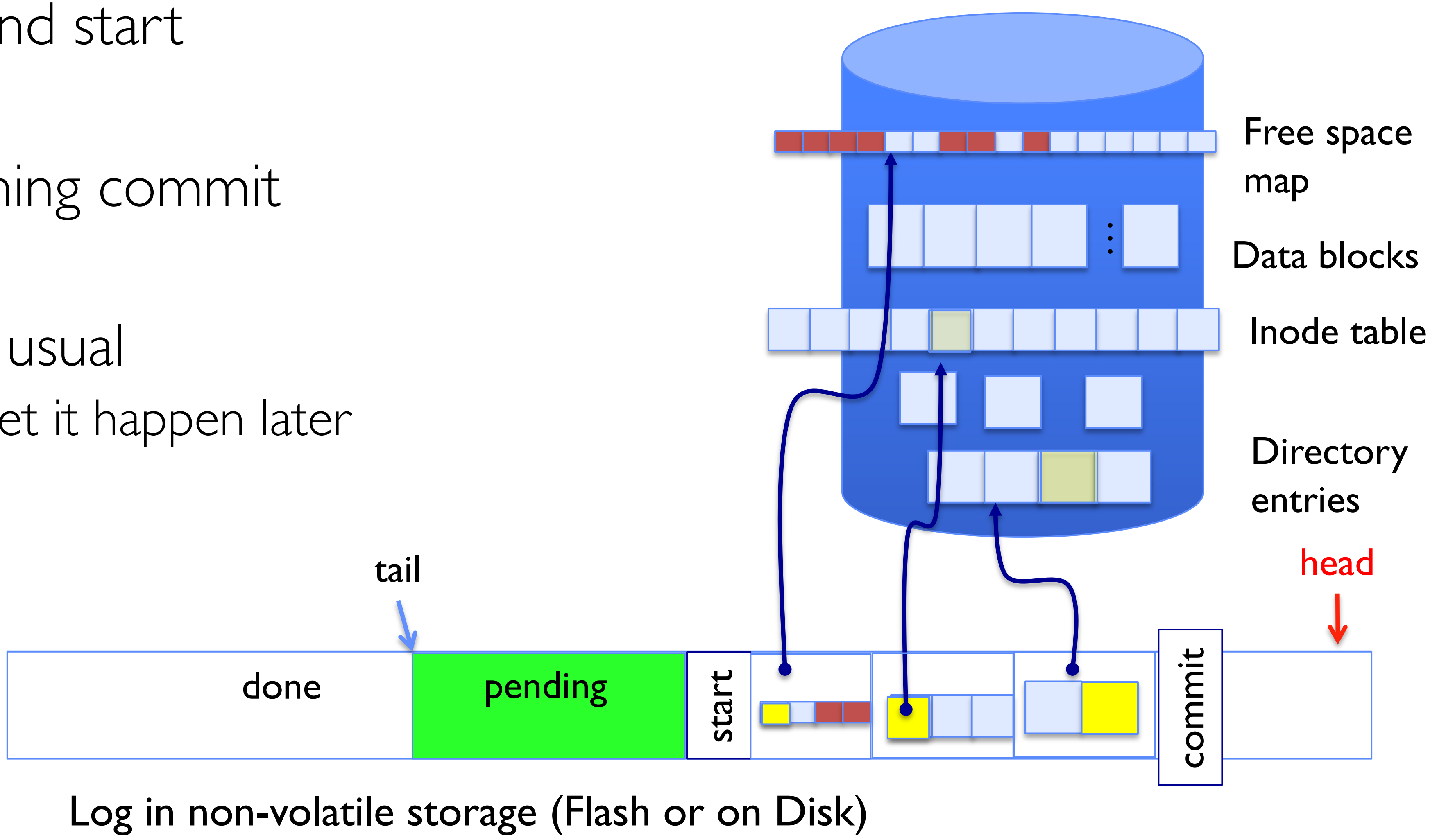
- Upon recovery scan the log
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



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



Implementation Details

- Deal with concurrent transactions
  - Must identify which transaction does a record belong to

- Repeated write-backs are OK
  - Works for idempotent (幂等) updates: "write 42 to each byte of sector 74"
  - Redo log systems do not permit non-idempotent records such as "add 42 to each byte in sector 74".

- Restarting recovery is OK
  - If another crash occurs during recovery

Implementation Details

- The performance of redo logging is not as bad as it looks like:
  - Log updates are sequential
  - Asynchronous write-back
    - ❑ Low latency for commit(); high throughput as updates can be batched
  - Group commit: combine a set of transaction commits into one log write
    - ❑ Amortize the cost of initiating the write (e.g., seek and rotational delays).
- New requests (e.g., reads) need to consult the log first to ensure the data consistency
  - Can be alleviated by caching
- Ordering is essential, as we must ensure:
  - A transaction's updates are on disk in the log before the commit is
  - The commit is on disk before any of the write-backs are
  - All of the write-backs are on disk before a transaction's log records are garbage collected.

Transactional File Systems

- Two ways to use transactions in file systems: journaling (日志) and logging

- Journaling: apply updates to the system's metadata via transactions
  - Microsoft's NTFS, Apple's HFS+, and Linux's XFS/JFS

- (Full) Logging: apply both metadata and data in transactions
  - Linux's ext3 and ext4 can be configured to use either journaling or logging

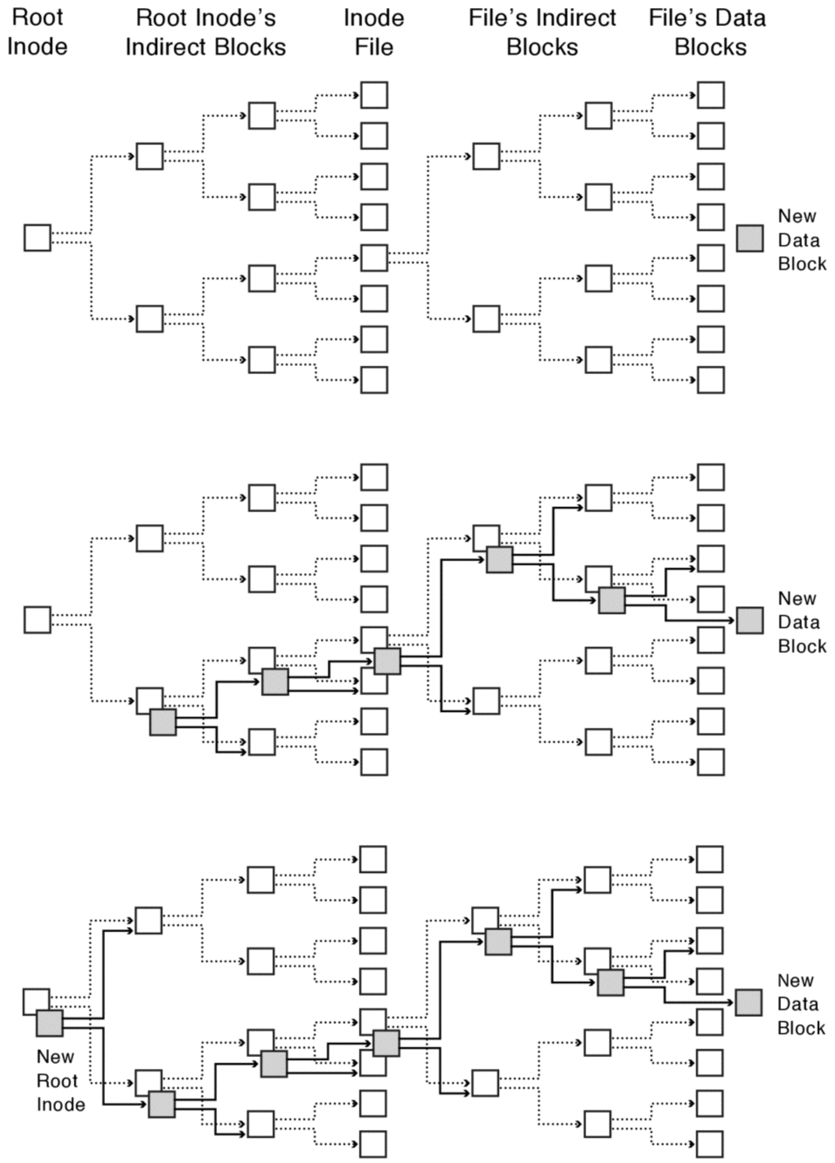
Journaling File Systems

- Applies updates to system metadata (inodes, bitmaps, directories, and indirect blocks) using transactions
  - So those critical data structures are always consistent

- Updates to non-directory files (i.e., user stuff) can be done in place (without logs), full logging optional
  - Avoids writing file contents twice
  - If a program using a journaling file system requires atomic multi-block updates, it needs to provide them itself

Copy-on-Write File System

- To update file system, write a new version of the file system containing the update
  - Never update in place
  - Reuse existing unchanged disk blocks
- Optimization: batch updates
  - Transform many small, random writes into large, sequential writes
- Approach taken in network file server appliances
  - NetApp's Write Anywhere File Layout (WAFL)
  - ZFS (Sun/Oracle) and OpenZFS

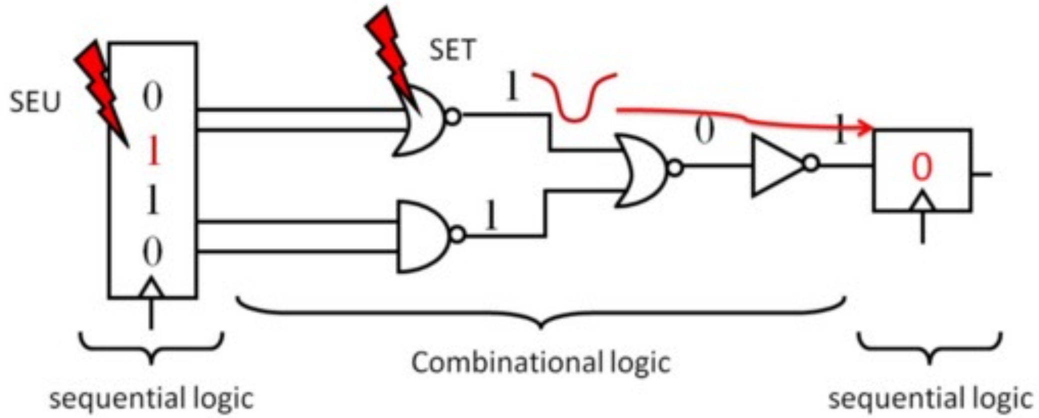


Goals for Today

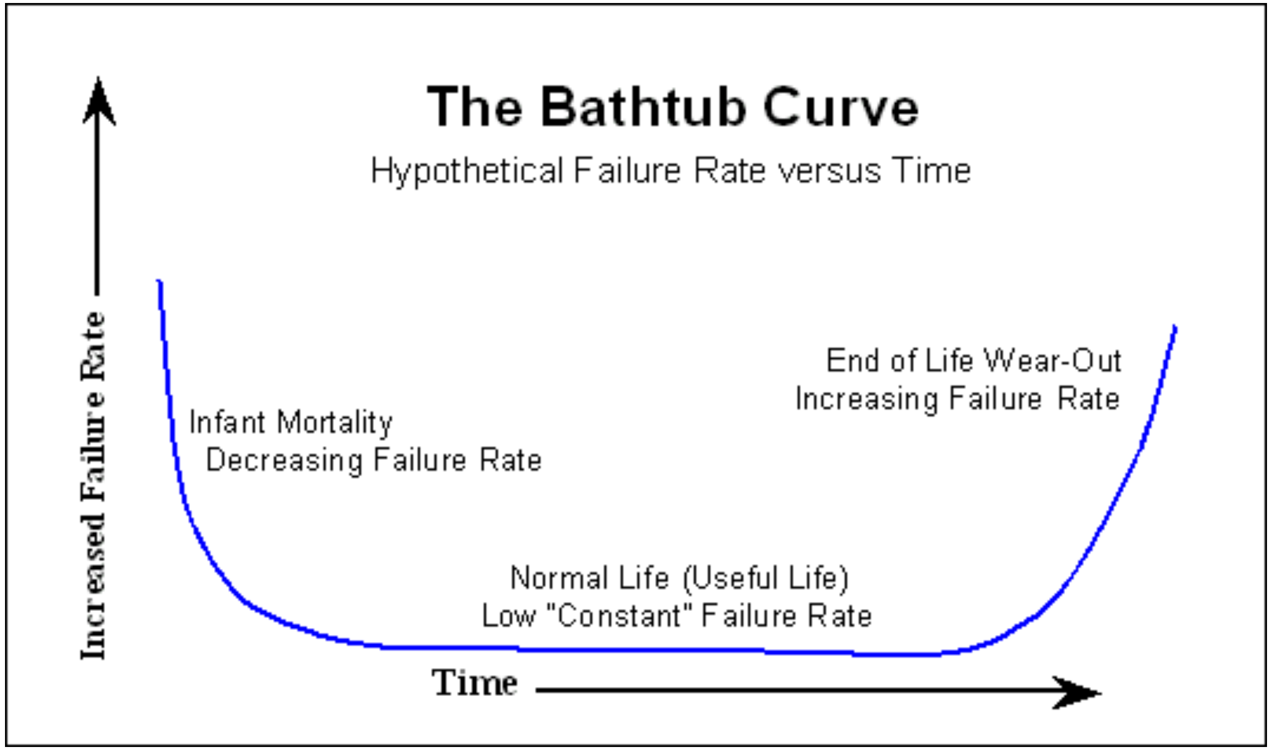
- Transactions for atomic updates
  - Redo Logging
- Redundancy for media failures
  - RAID

Storage Devices Failure

- **Sector and page failure**: one or more individual sectors of a disk are lost, but the rest of the disk continues to operate correctly
- **Full disk failure:** a device stops being able to service reads or writes to all sectors



Single-event upset (单粒子翻转)

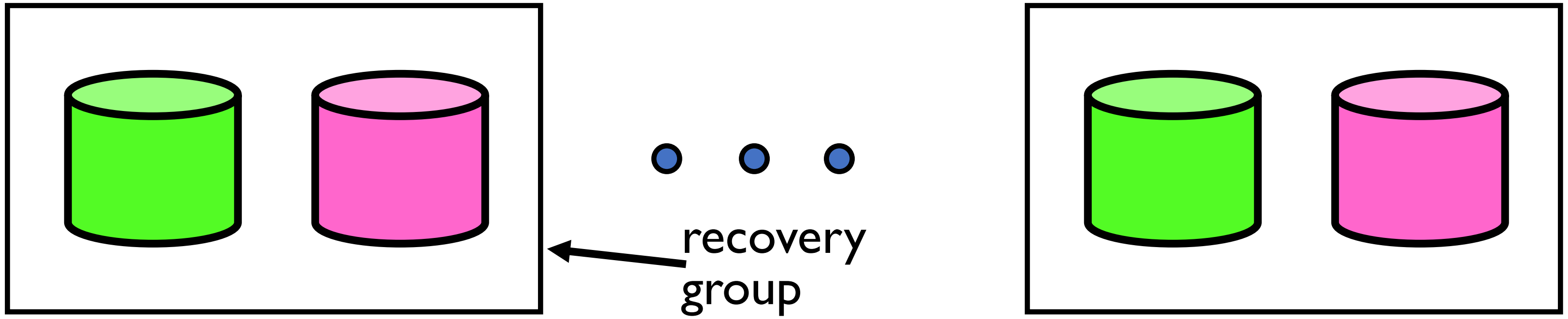


RAID: Redundant Arrays of Inexpensive Disks

- Invented by David Patterson, Garth A. Gibson, and Randy Katz here at UCB in 1987
- Data stored on multiple disks (redundancy)
- 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)
- 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

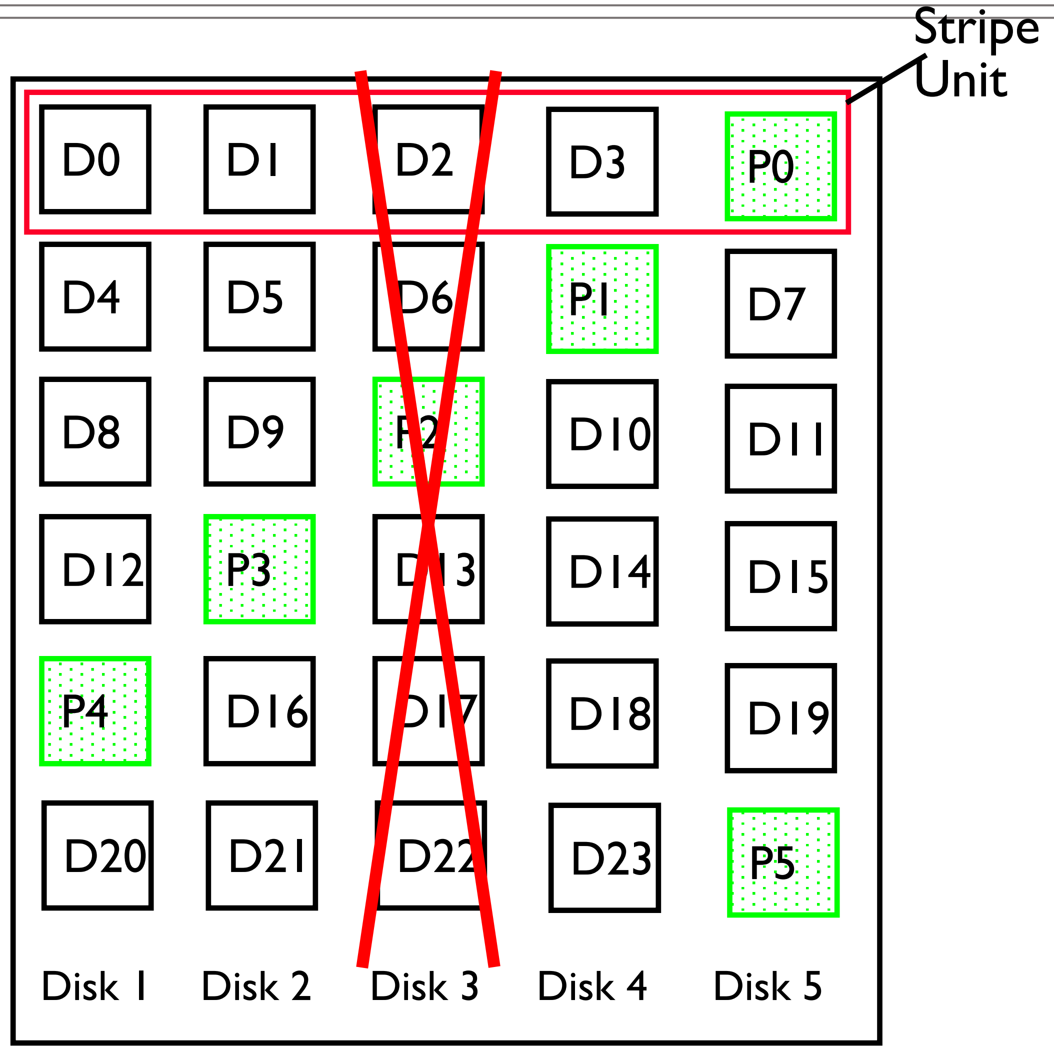


Magic XOR (异或)

- XOR (^), or eXclusive OR, is a bitwise operator that returns true (1) for odd frequencies of 1. The XOR truth table is as follows:
  - 1 ^ 1 = 0
  - 1 ^ 0 = 1
  - 0 ^ 1 = 1
  - 0 ^ 0 = 0
- XOR is commutative.
  - a^b = b^a.
- XOR is associative.
  - a^(b^c) = (a^b)^c = (a^c)^b.
- XOR is self-inverse.
  - Any number XOR'ed with itself evaluates to 0.
- a^a = 0.
  - 0 is the identity element for XOR.
- This means, any number XOR'ed with 0 remains unchanged.
  - a^0 = a.

RAID 5+: High I/O Rate Parity

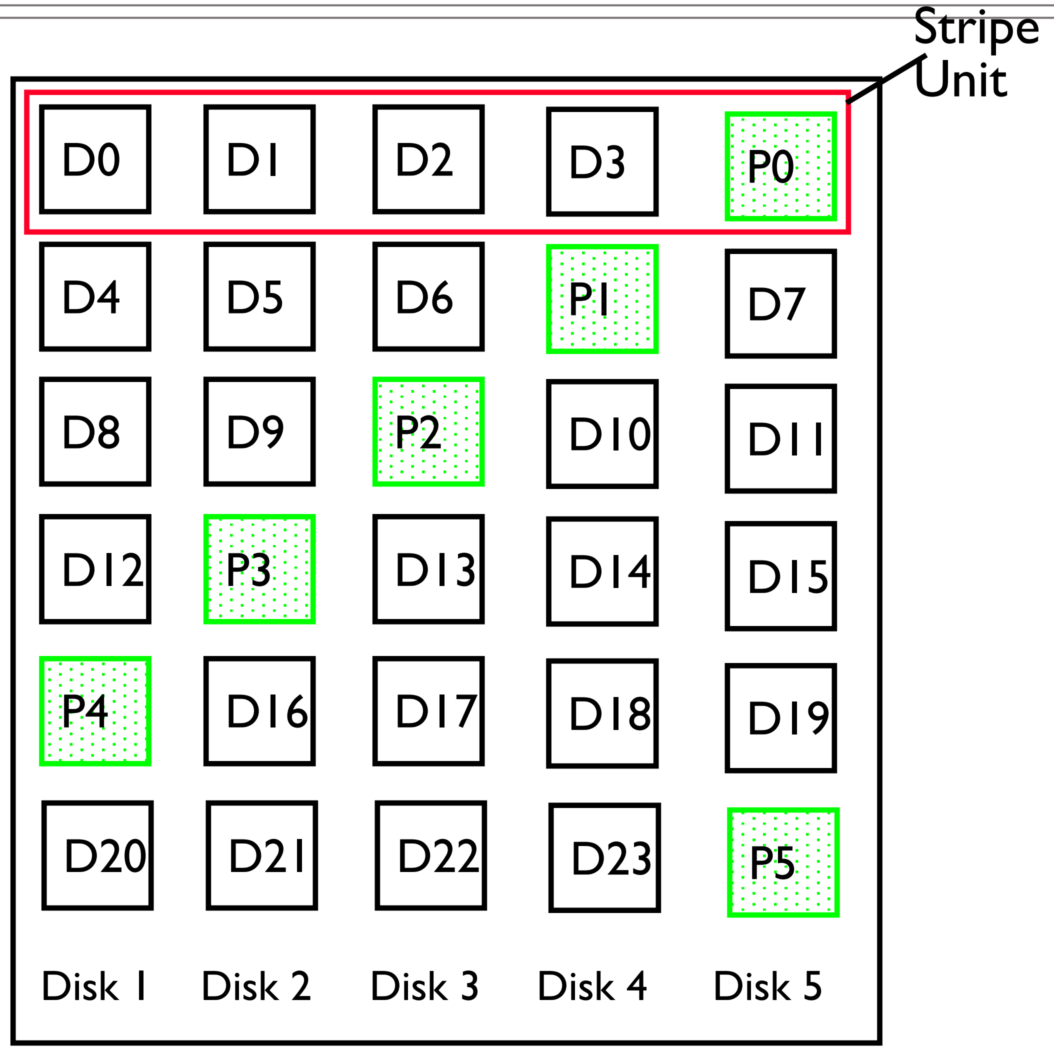
- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing (异或) data blocks in stripe
  - $P0 = D0 \oplus D1 \oplus D2 \oplus D3$
  - Can destroy any one disk and still reconstruct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



RAID 5+: High I/O Rate Parity

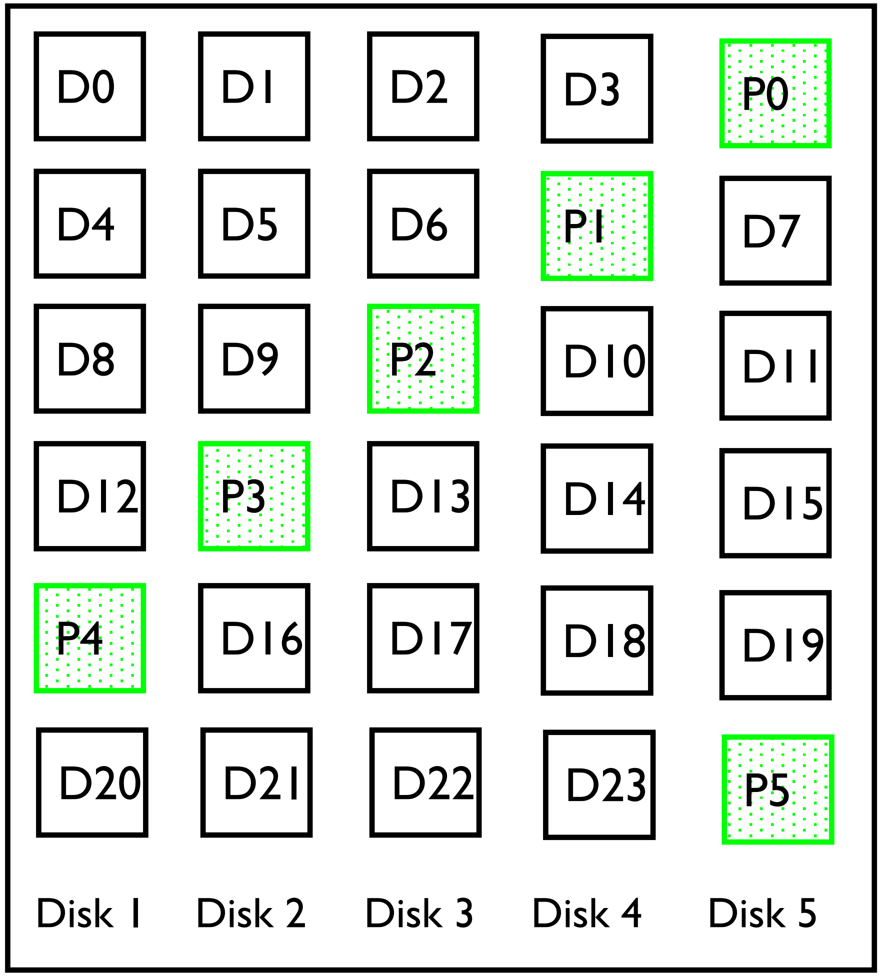
- Rotating parity (奇偶校验)
  - The parity needs to be updated more often than normal data blocks.
- Striping data
  - Balance parallelism vs. sequential access efficiency

- RAID 5 can recover the failed disk only if (i) only one disk fails and (ii) the failed disk is known.



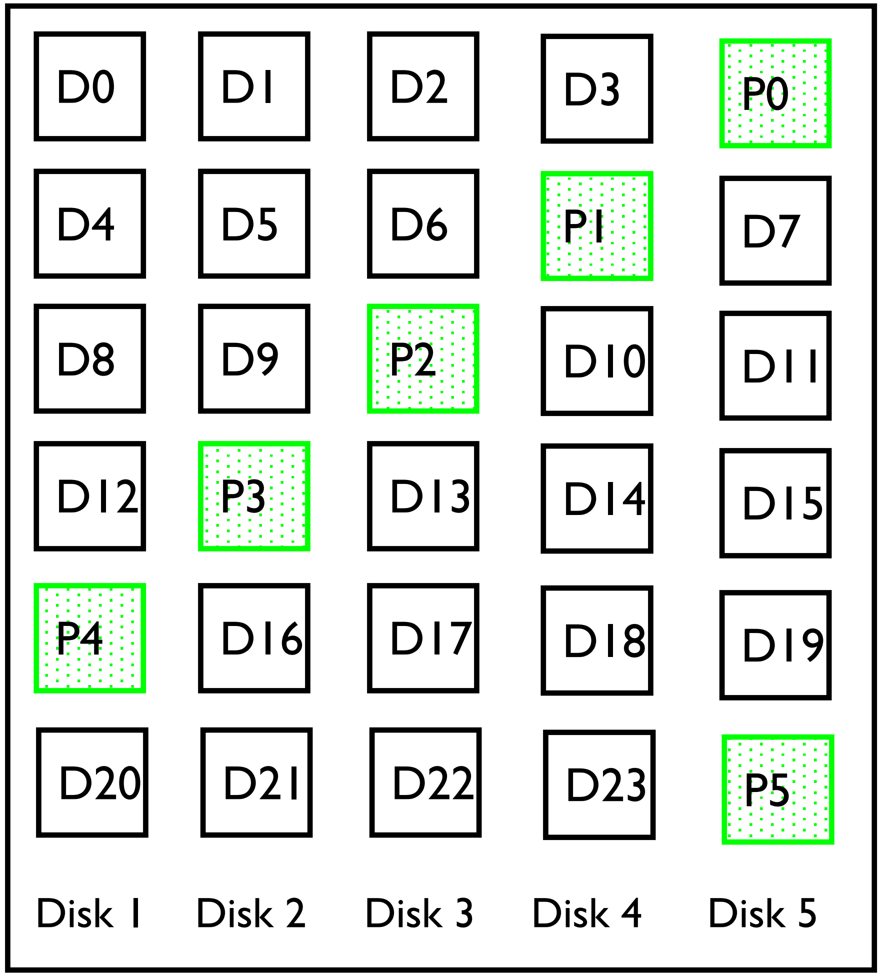
RAID 5+: High I/O Rate Parity

- What I/O operations would occur if we want to update D21 in this figure?



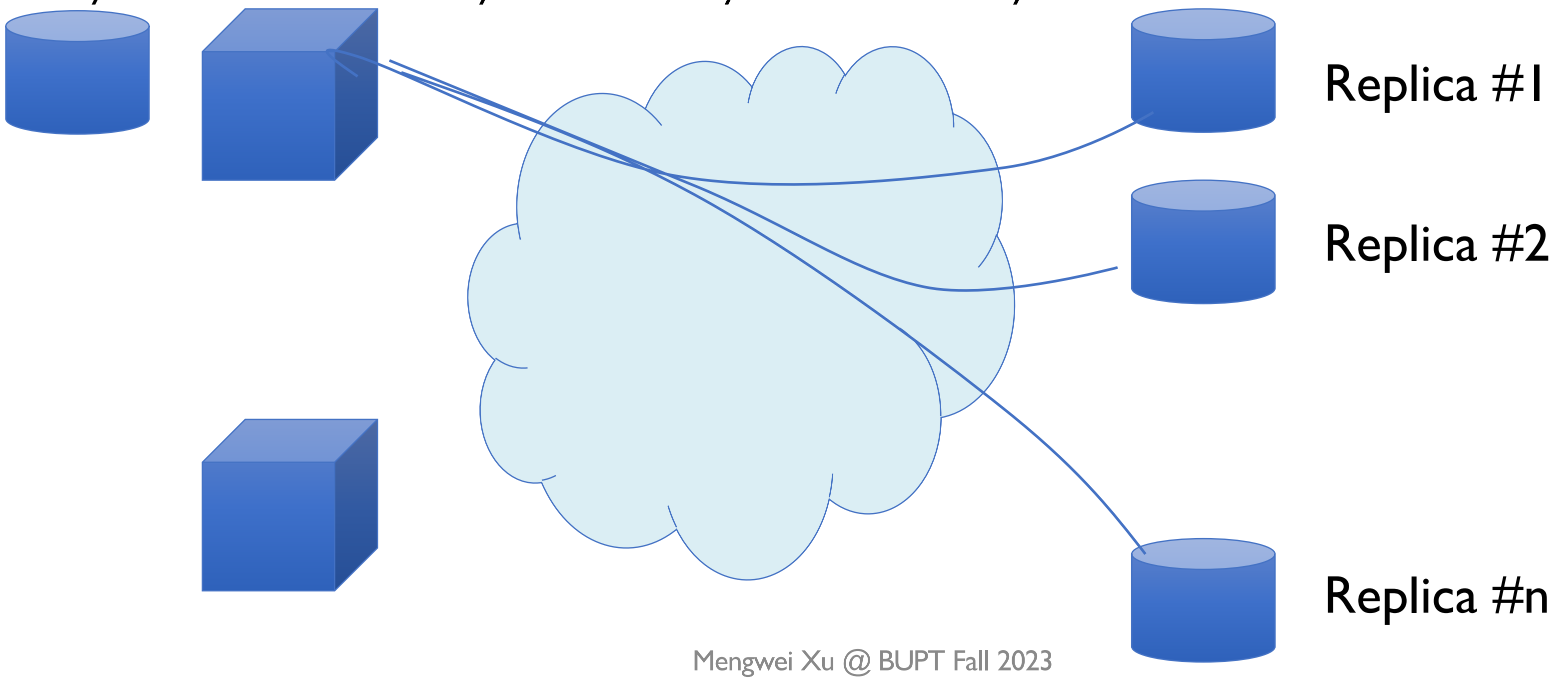
RAID 5+: High I/O Rate Parity

- What I/O operations would occur if we want to update D21 in this figure?
  - Read D21(old)
  - Read P5(old)
  - Compute tmp=P5(old)⊕D21(old)
  - Compute P5(new)=tmp ⊕D21(new)
  - Write D21(new)
  - Write P5(new)



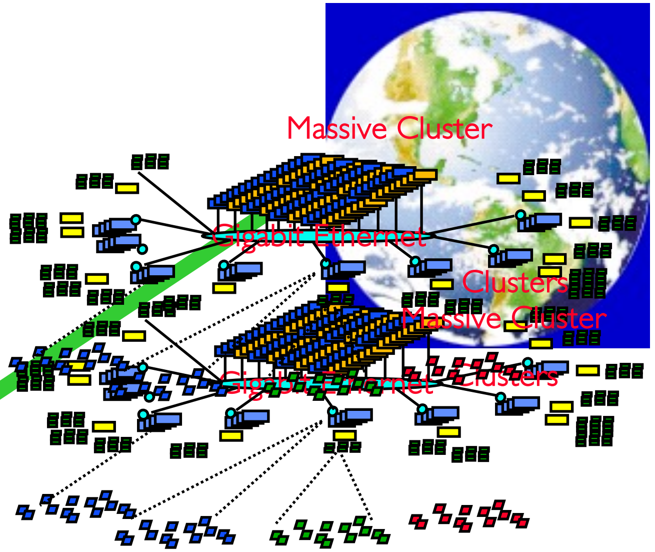
Higher Durability/Reliability through Geographic Replication

- Highly durable – hard to destroy all copies
- Highly available for reads – read any copy
- 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



Societal Scale Information Systems

- The world is a large distributed system
  - Microprocessors in everything
  - Vast infrastructure behind them



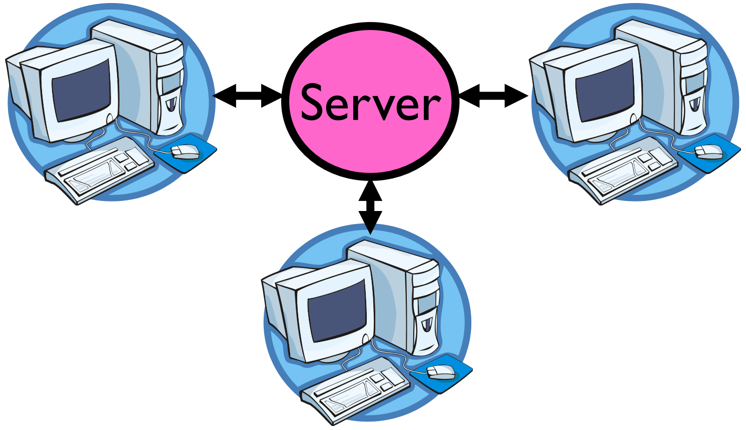
Scalable, Reliable, Secure Services

Internet Connectivity

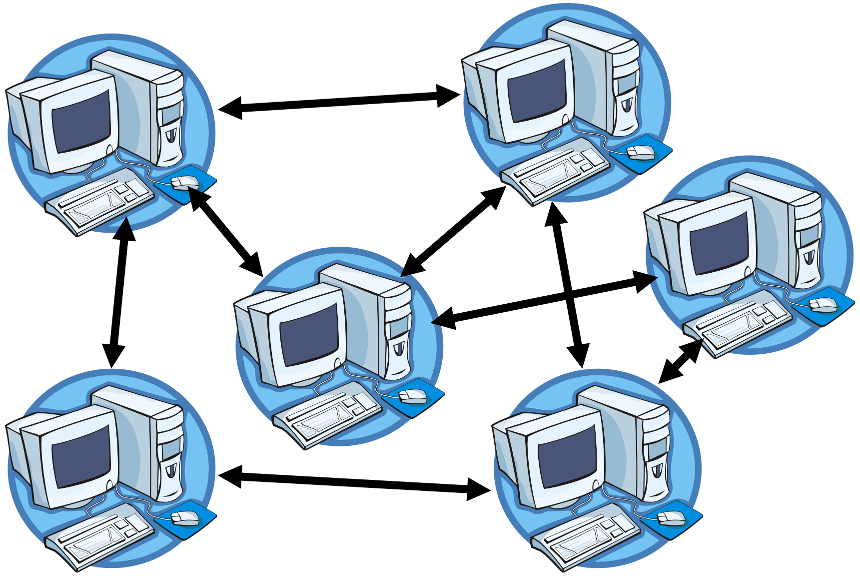
MEMS for Sensor Nets

Databases
Information Collection
Remote Storage
Online Games
Commerce
…

Centralized vs Distributed Systems



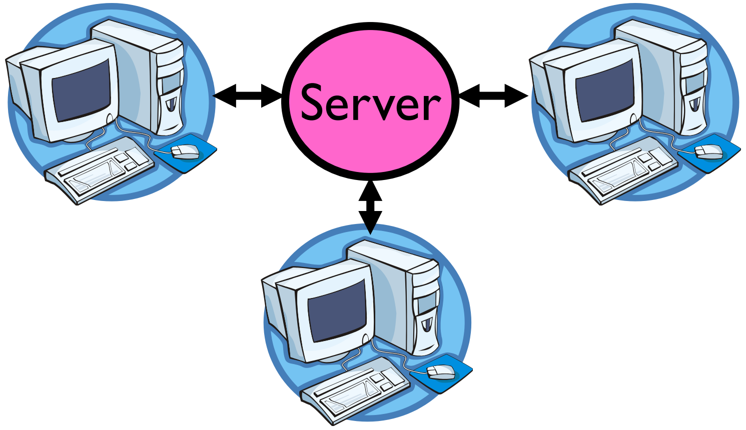
Client/Server Model



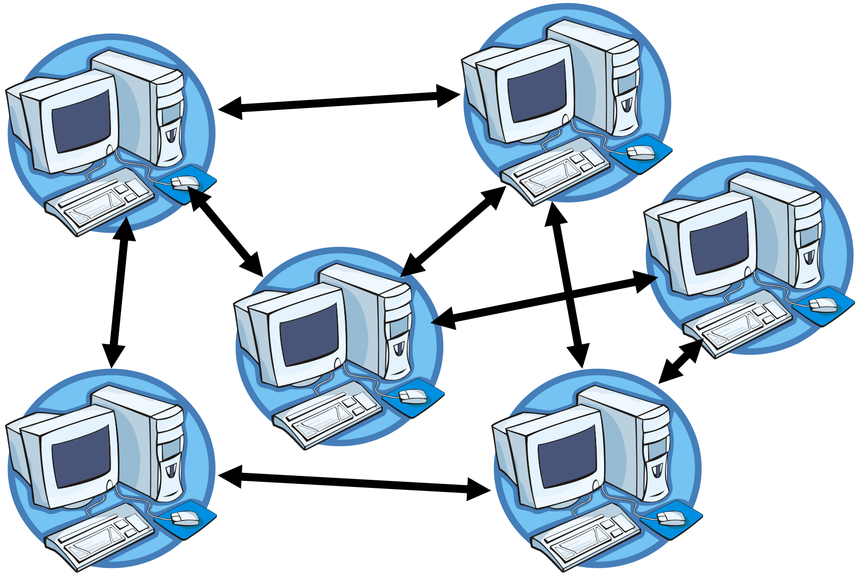
Peer-to-Peer Model

- Centralized System: System in which major functions are performed by a single physical computer
  - Originally, everything on single computer
  - Later: client/server model

Centralized vs Distributed Systems



Client/Server Model



Peer-to-Peer Model

- Distributed System: physically separate computers working together on some 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
  - Easier to add power 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 where I can't do work because some machine I've never heard of isn't working!"
  - 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 (using only a network)
  - What would be easy in a centralized system becomes a lot more difficult

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

Homework-1

- The FastFile file system uses an inode array to organize the files on disk. Each inode consists of a user id (2 bytes), three time stamps (4 bytes each), protection bits (2 bytes), a reference count (2 byte), a file type (2 bytes) and the size (4 bytes). Additionally, the inode contains 13 direct indexes, 1 index to a 1st-level index table, 1 index to a 2nd-level index table, and 1 index to a 3rd level index table. The file system also stores the first 436 bytes of each file in the inode.
  - Assume a disk sector is 512 bytes, and assume that any auxilliary index table takes up an entire sector, what is the maximum size for a file in this system.
  - Is there any benefit for including the first 436 bytes of the file in the inode?

Homework-2

- When user tries to write a file, the file system needs to detect if that file is a directory so that it can restrict writes to maintain the directory's internal consistency. Given a file's name, how would you design a file system to keep track of whether each file is a regular file or a directory?
    - In FAT
    - In FFS
    - In NTFS

Homework-3

- Suppose a variation of FFS includes in each inode 12 direct, 1 indirect, 1 double indirect, 2 triple indirect, and 1 quadruple indirect pointers. Assuming 6 KB blocks and 6-byte pointers.
  - What is the largest file that can be accessed with direct pointers only?
  - What is the largest file that can be accessed in total?

Homework-4

- Consider a disk queue holding requests to the following cylinders in the listed order: 116, 22, 3, 11, 75, 185, 100, 87. Using the elevator scheduling algorithm, what is the order that the requests are serviced, assuming the disk head is at cylinder 88 and moving upward through the cylinders?

Homework-5

- Search for how different RAID versions (at least 5) work differently and list a table to compare them.