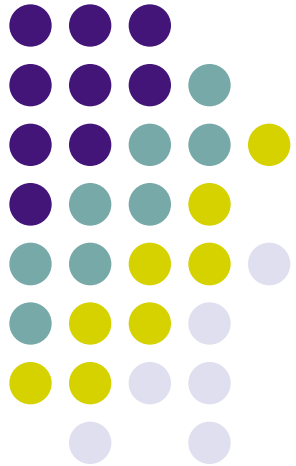
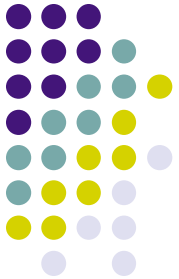
RAID, Distributed FS Intro

ECE469 April 13

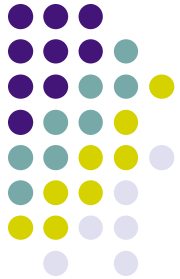
Yiying Zhang





- Reading: Chapter 12
- Lab 5 out, lab 4 due this Friday
- Quiz3 next Tuesday
- No class next Thursday
- April 25th last lecture
- Final May 3rd

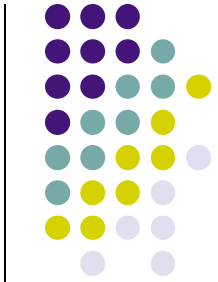
[lec24] File system reliability



- Loss of data in a file system can have catastrophic effect
  - How does it compare to hardware (DRAM) failure?
  - Need to ensure safety against data loss

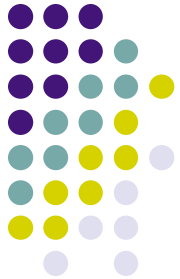
- Three threats:
  - Accidental or malicious deletion of data → backup
  - Media (disk) failure → data replication (e.g., RAID)
  - System crash during file system modifications → consistency

[lec24] 1. Backup



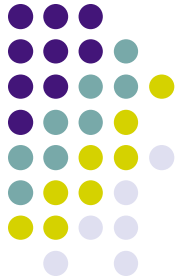
- Copy entire file system onto low-cost media (tape), at regular intervals (e.g. once a day).
  - Backup storage (cold storage)
- In the event of a disk failure, replace disk and restore from backup media
- Amount of loss is limited to modifications occurred since last backup

[lec24] 2. Data Replication



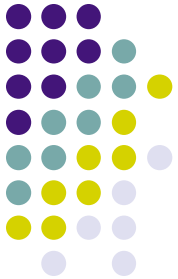
- Full replication
  - Mirroring across disks
  - Full replication to different machines (more next week)
- RAID (today)
- Erasure Coding
  - Like RAID, use parity, but saves more space

[lec24] 3. Crash Recovery



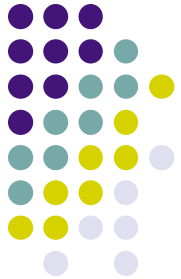
- After a system crash in the middle of a file system operation, file system metadata may be in an *inconsistent state*

[lec24] ext3 journaling



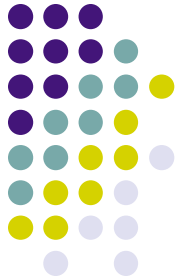
- Journal location
  - EITHER on a separate device partition
  - OR just a “special” file within ext2
- Three separate modes of operation:
  - Data: All data is journaled
  - Ordered, Writeback: Just metadata is journaled
- First focus: Data journaling mode

[lec24] Transactions in ext3 Data Journaling Mode



- Same example: Update Inode (I), Bitmap (B), Data (D)
- First, write to journal:
  - Transaction begin (Tx begin)
  - Transaction descriptor (info about this Tx)
  - I, B, and D blocks (in this example)
  - Transaction end (Tx end)
- Then, "checkpoint" data to fixed ext2 structures
  - Copy I, B, and D to their fixed file system locations
- Finally, free Tx in journal
  - Journal is fixed-sized circular buffer, entries must be periodically freed

[lec24] What if there's a Crash?

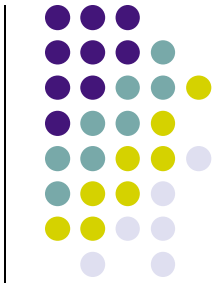


- Recovery: Go through log and "redo" operations that have been successfully committed to log

- What if …
  - Tx begin but not Tx end in log?
  - Tx begin through Tx end are in log, but I, B, and D have not yet been checkpointed?
  - What if Tx is in log, I, B, D have been checkpointed, but Tx has not been freed from log?

- Performance? (As compared to fsck?)

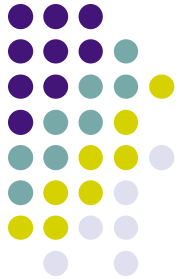
[lec24] Problem with Data Journaling



- Data journaling: Lots of extra writes
  - All data committed to disk twice (once in journal, once to final location)
- Overkill if only goal is to keep metadata consistent
  - Why is this sometimes OK?

- Instead, use writeback mode
  - Just journals metadata
  - Data is not journaled. Writes data to final location directly
  - Better performance than data journaling (data written once)
  - The contents might be written at any time (before or after the journal is updated)
- Problems?
  - If a crash happens, metadata can point to old or even garbage data!

[lec24] Ext3 ordered journaling mode

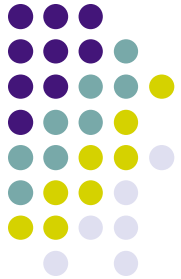


- How to order data block write w.r.t. journal (metadata) writes?

- Ordered journaling mode
  - Only metadata is journaled, file contents are not (like writeback mode)
  - But file contents guaranteed to be written to disk before associated metadata is marked as committed in the journal
  - Default ext3 journaling mode

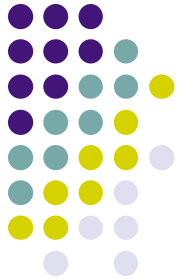
  - What happens when crash happens?
    - Metadata will only point to correct data (no stale data can be reached after reboot).
    - But there may be data that is not pointed to by any metadata.
    - How is this better than writeback in terms of consistency guarantees?

Complication: Disk/SSD Scheduling



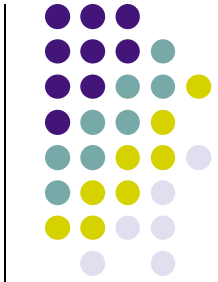
- Problem: Low-levels of I/O subsystem in OS and even the disk/RAID itself may reorder requests
- How does this affect Tx management?
  - Where is it OK to issue writes in parallel?
    - Tx begin
    - Tx info
    - I, B, D
    - Tx end
    - Checkpoint: I, B, D copied to final destinations
    - Tx freed in journal

Complication: Disk/SSD Buffering



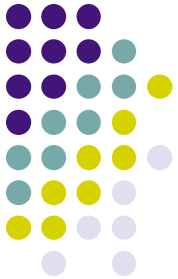
- Problem: Disks (SSDs) have internal memory to buffer writes. When the OS writes to disk, it does not necessarily mean that the data is written to persistent media
- How does this affect Tx management?
  - Tx begin
  - Tx info
  - I, B, D
  - Tx end
  - Checkpoint: I, B, D copied to final destinations
  - Tx freed in journal

Conclusions



- Journaling
  - Almost all modern file systems use journaling to reduce recovery time during startup (e.g., Linux ext3, ReiserFS, SGI XFS, IBM JFS, NTFS)
  - Simple idea: Use write-ahead log to record some info about what you are going to do before doing it
  - Turns multi-write update sequence into a single atomic update (“all or nothing”)
  - Some performance overhead: Extra writes to journal
    - Worth the cost?

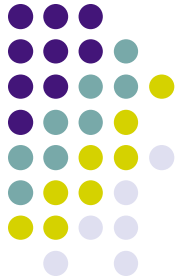
Disk Observations



- Getting first byte from disk read is slow
  - high ***latency***
- Peak disk bandwidth good, but rarely achieved

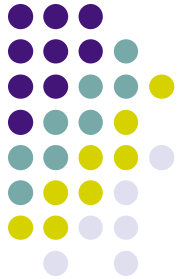
- Towards mitigate disk performance impact
  - Disk caches (read-ahead and write buffer)
  - Move some disk data into main memory – file caching
  - Disk scheduling
    - There are often multiple disk requests outstanding
    - Schedule requests to shorten seeks!
  - *What else can we try?*
    - *Adding multiple disk arms to the disk?*

RAID



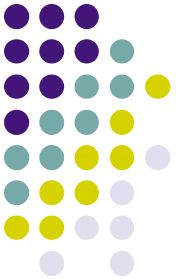
- Two motivations
  - (in the past) Operating in parallel can increase disk throughput
    - RAID = Redundant Array of Inexpensive Disks
  - (today) Redundancy can increase reliability
    - RAID = Redundant Array of Independent Disks

RAID -- Two main ideas (1)



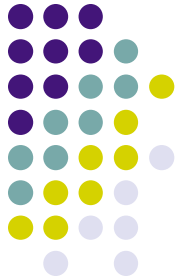
- Parallel reading (striping) (for performance)
  - Splitting bits of a byte across multiple disks
  - 8 disks (bit-level striping)
    - Logically acts like single disk with sector size X 8 and access time / 8
    - Reduce the response time of large access (e.g. 1 4K block)
  - Alternatively, block-level striping
    - Increases the throughput of multiple small accesses (e.g. 8 512-byte blocks)

RAID -- Two main ideas (2)



- Mirroring or shadowing (for reliability)
  - Local disk consists of 2 physical disks in parallel
  - Every write performed on both disks
    - Can read from either disk
    - Probability of both fail at the same time?

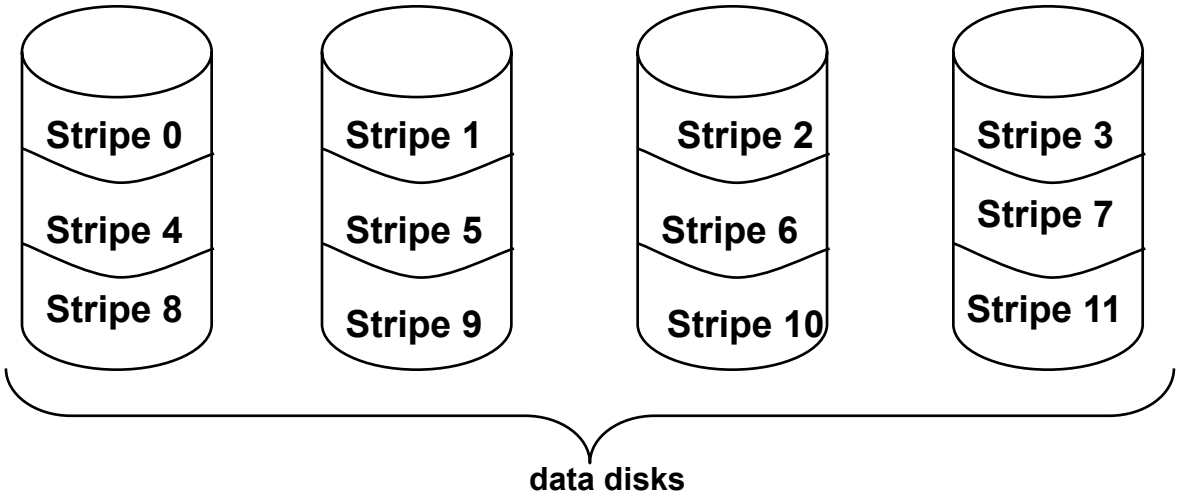
RAID – combing the two ideas!



- Mirroring gives reliability, but expensive
- Striping gives high data-transfer rate, but not reliability
- Challenge: can we provide redundancy at low cost?

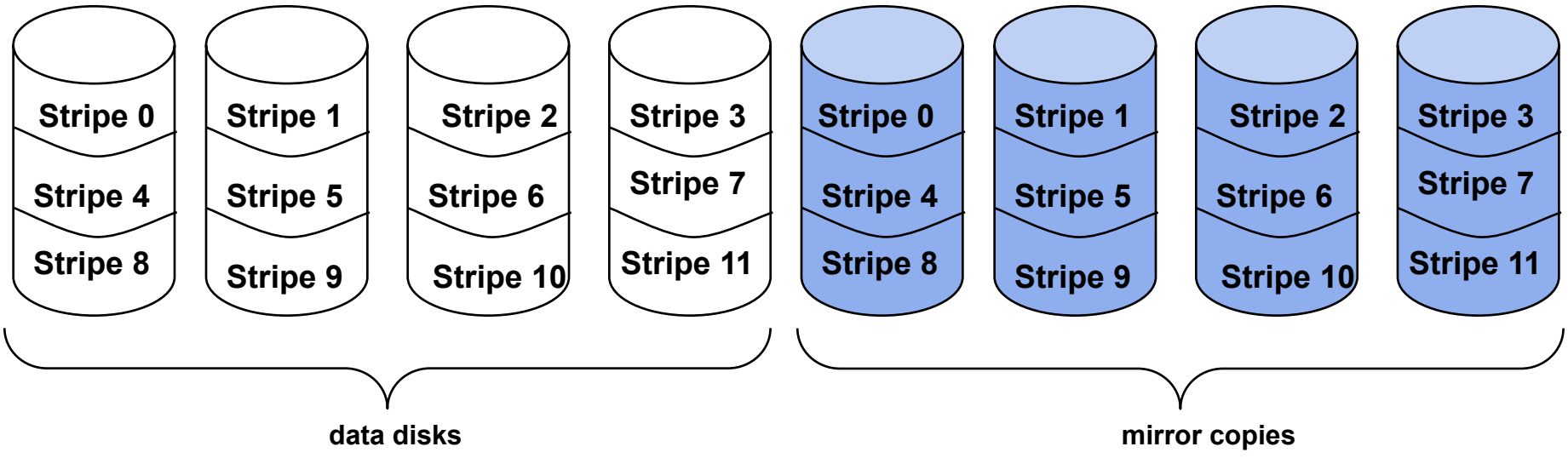
Raid Level 0

- Level 0 is non-redundant disk array
- Files are Striped across disks, no redundant info
- High read throughput
- Best write throughput among RAID levels (no redundant info to write)
- Any disk failure results in data loss
  - What's the MTTF (mean time to failure) of the whole system?

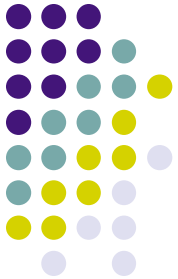


Raid Level 1

- Mirrored Disks
- Data is written to two places
  - On failure, just use surviving disk
- On read, choose fastest to read
  - Write performance is same as single drive, read performance is 2x better
- Expensive

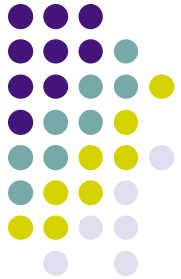


Parity and Hamming Codes



- What do you need to do in order to detect and correct a one-bit error ?
  - Suppose you have a binary number, represented as a collection of bits: <b3, b2, b1, b0>, e.g. 0110
- Detection is easy
- Parity:
  - Count the number of bits that are on, see if it's odd or even
    - EVEN parity is 0 if the number of 1 bits is even
  - Parity(<b3, b2, b1, b0 >) = P0 = b0 ⊗ b1 ⊗ b2 ⊗ b3
  - Parity(<b3, b2, b1, b0, p0>) = 0 if all bits are intact
  - Parity(0110) = 0, Parity(01100) = 0
  - Parity(11100) = 1 => ERROR!
  - Parity can detect a single bit error, but can't tell you which of the bits got flipped

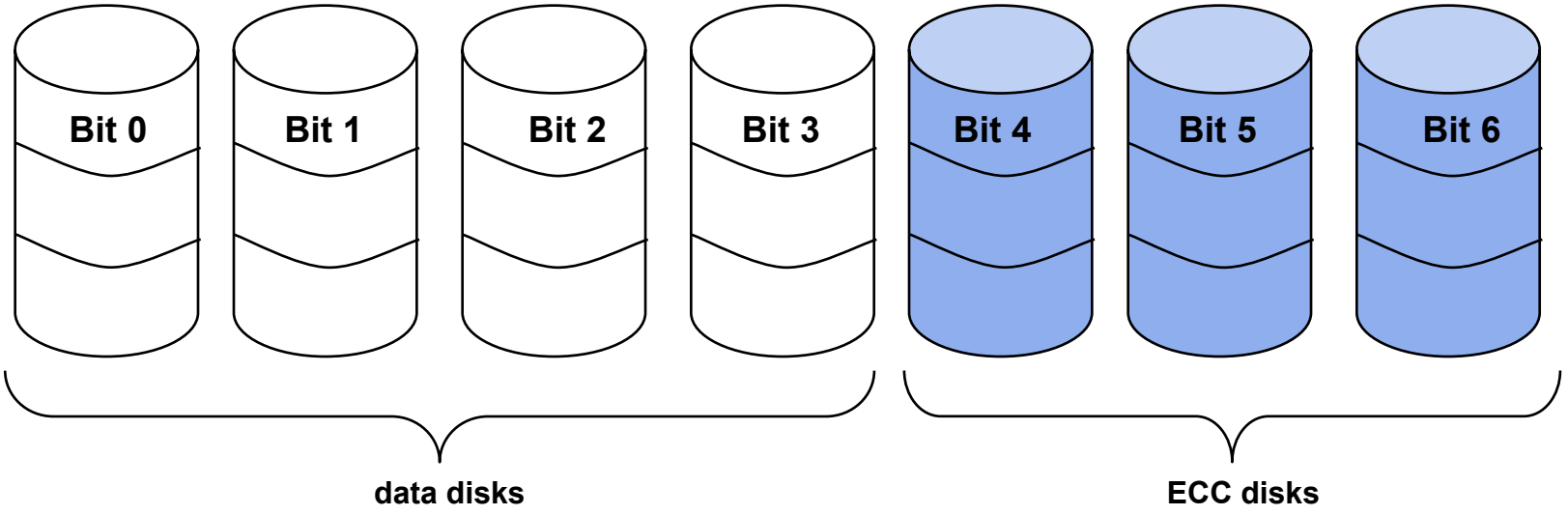
Parity and Hamming Code



- Detection and correction require more work
- Hamming codes can detect double bit errors and detect & correct single bit errors
  - Details omitted in this course, can be found in other courses

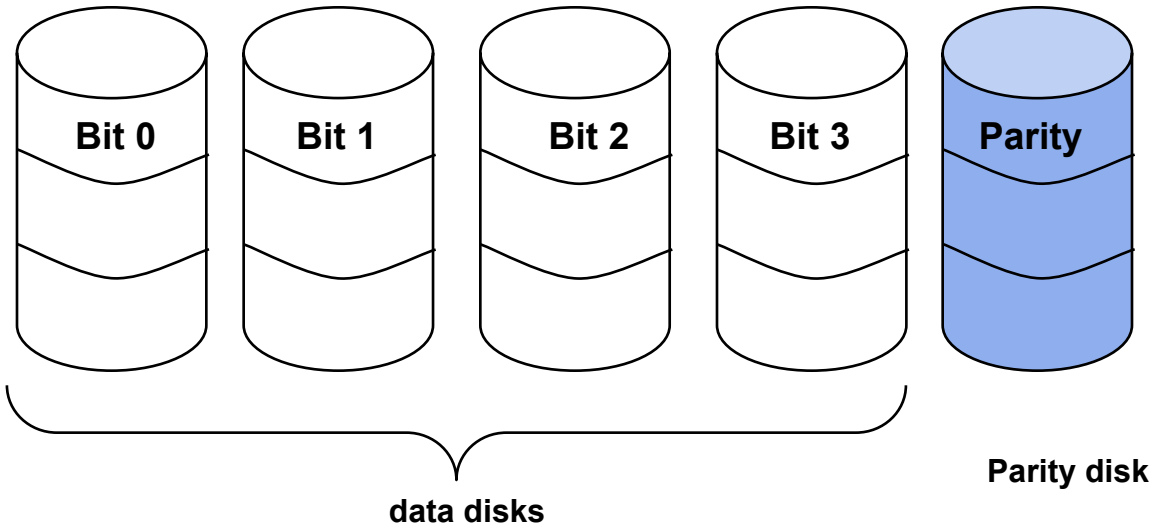
Raid Level 2

- Bit-level Striping with Hamming (ECC) codes for error correction
- All 7 disk arms are synchronized and move in unison
- Complicated controller
- Single access at a time
- Tolerates only one error, but with no performance degradation
- Not used in real world



Raid Level 3

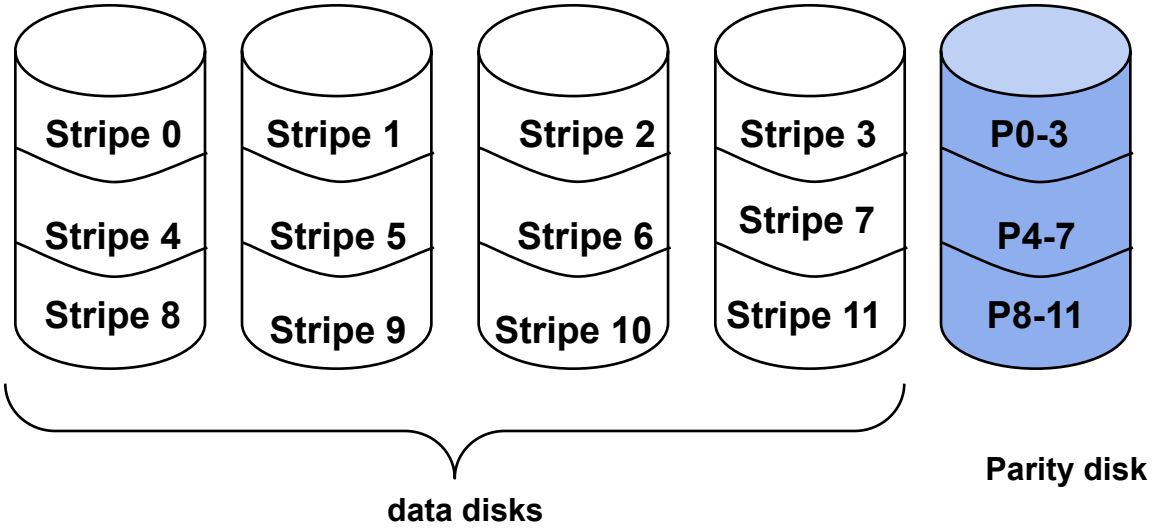
- Use a parity disk
  - Each bit on the parity disk is a parity function of the corresponding bits on all the other disks
- A read accesses all the data disks
- A write accesses all data disks plus the parity disk
- On disk failure, read remaining disks plus parity disk to compute the missing data



Single parity disk can be used to detect and recover errors

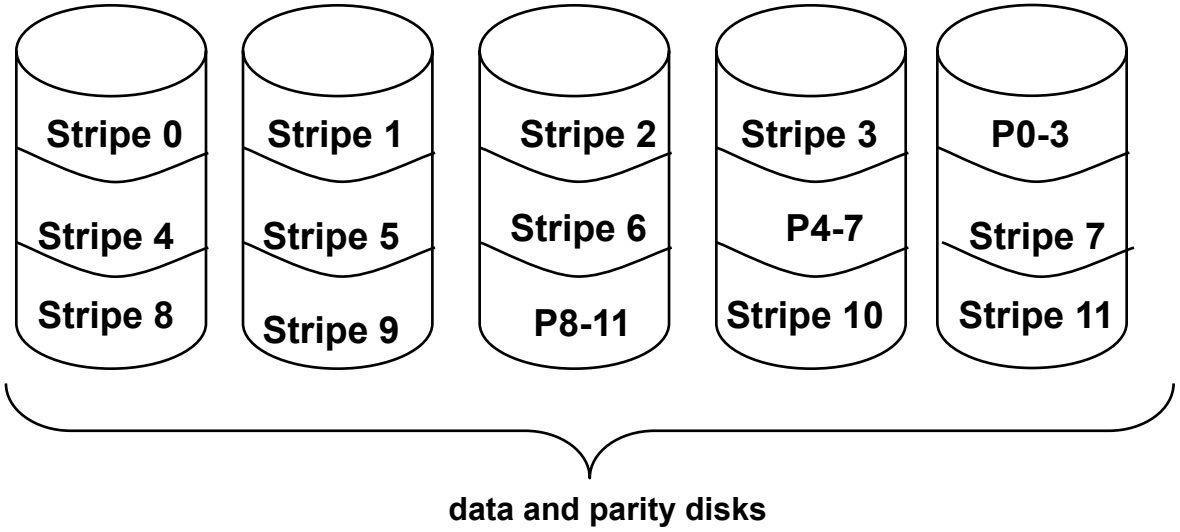
Raid Level 4

- Combines Level 0 and 3 – block-level parity with Stripes
- Lower transfer rate for each block (by single disk)
- Higher overall rate (many small files, or a large file)
- Large writes → parity bits can be written in parallel
- Small writes → 2 reads + 2 writes !
- Heavy load on the parity disk



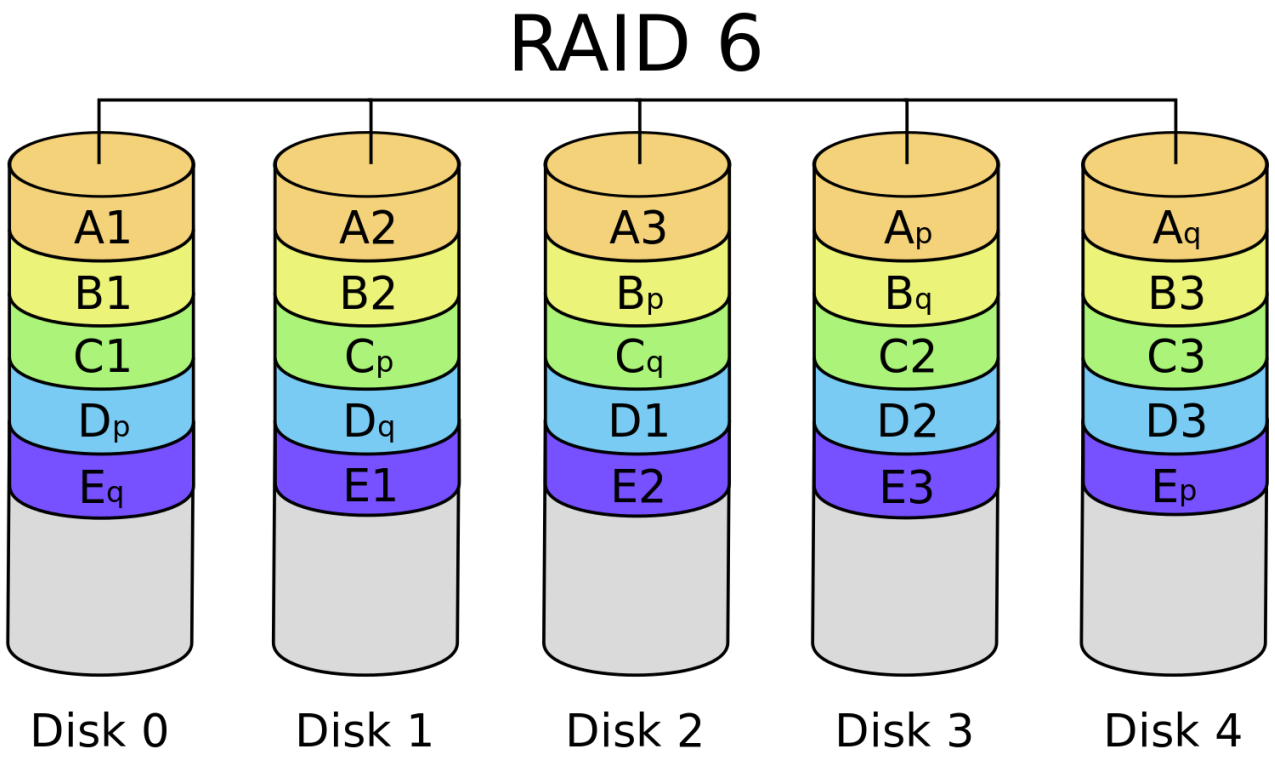
Raid Level 5

- Block Interleaved Distributed Parity
- Like parity scheme, but distribute the parity info over all disks (as well as data over all disks)
- Better read performance, large write performance
  - No single disk as performance bottleneck

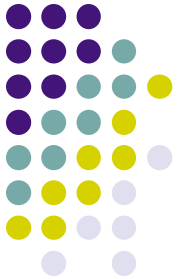


RAID 6

- Level 5 with an extra parity bit
- Can tolerate two failures
  - What are the odds of having two concurrent failures ?
- May outperform Level-5 on reads, slower on writes

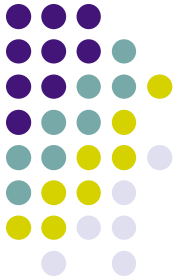


RAID Implementation



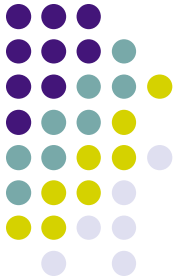
- Typically in hardware
  - Special-purpose RAID controller (PCI card)
  - Manages disks
  - Performs parity calculation
- Can be in software (by OS)
  - Can be fast
  - At the cost of CPU time

Andrew Ng



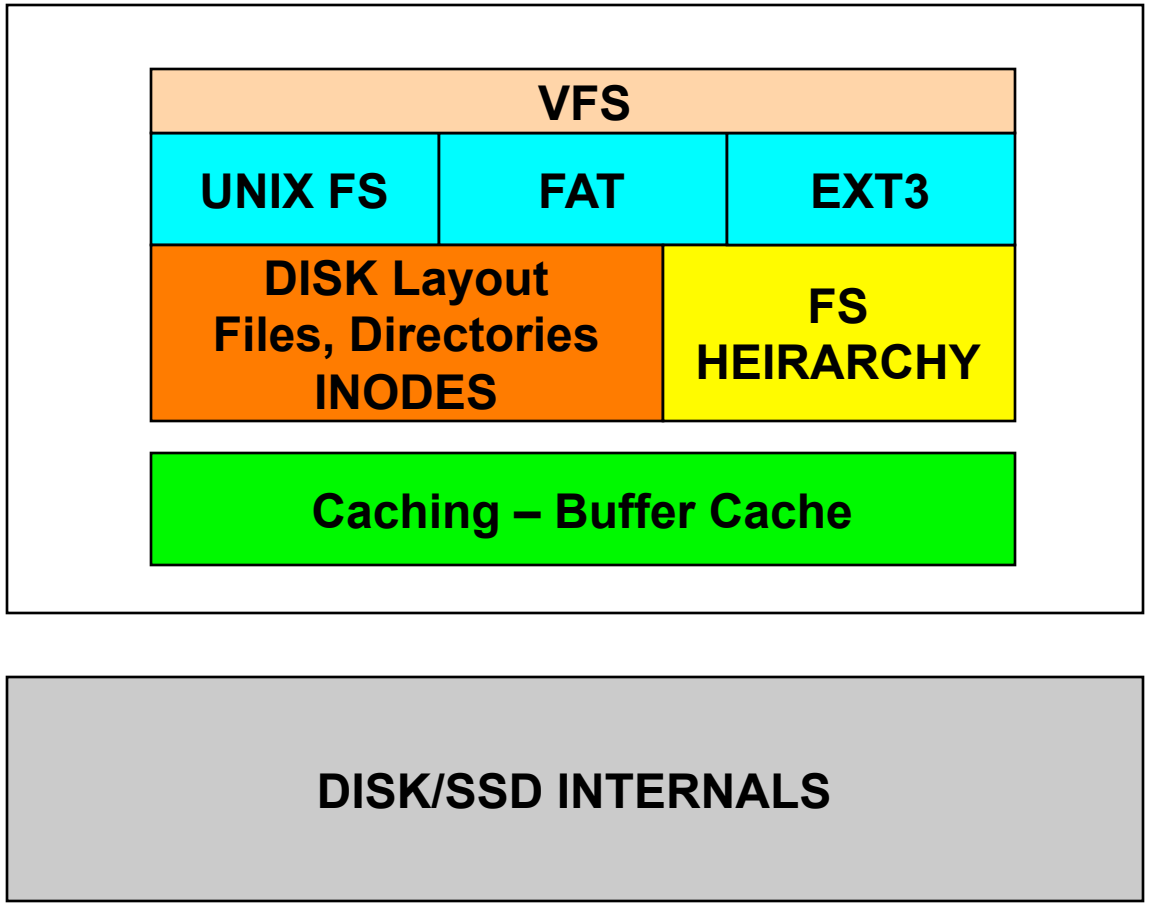
- Deep Learning
- Prof at Stanford
- Chief Scientist at Baidu leading deep learning group (recently quit)

Michael Jordan



- Machine Learning
- Berkeley

FS topics we have covered



DISK/SSD INTERNALS

FS Reliability

Crash Recovery

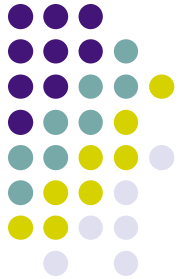
Journaling

RAID

Distributed File System (DFS)

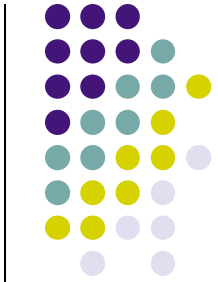
Network File System (NFS)

Distributed Systems



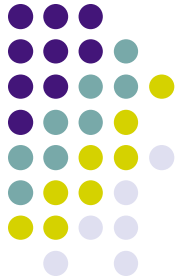
- Collection of computer nodes
- Connected by a network
- Running software that manages the distributed set of computers
  - Nodes work together to achieve a common goal (e.g., processing big data)
  - Nodes communicate and coordinate their actions by passing messages

Examples



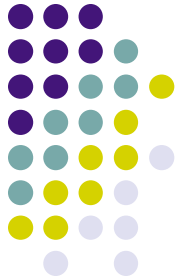
- Web 2.0 / Online services
  - Search, Social networks, Blogs, Wikis, Video Sharing
- Distributed parallel data processing
  - Hadoop, Spark
- Big data analysis
- Distributed graphs
- Banking
- Distributed supercomputing (Grid computing)
- Datacenter systems
- Peer-to-peer systems (e.g., BitTorrent)
- *Distributed storage/file systems*

Distributed systems



- Key differences from centralized systems
  - No physically shared memory
  - Communication: delays, unreliable
  - No common clock
  - Independent node failure modes
    - In a large-scale system, it's certain that something will fail!
  - Hardware/software heterogeneity

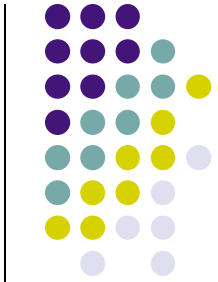
Distributed systems (cont)



- Need to revisit many OS issues
  - Distributed synchronization
  - Distributed deadlock detection
  - Distributed memory management
  - Distributed file systems

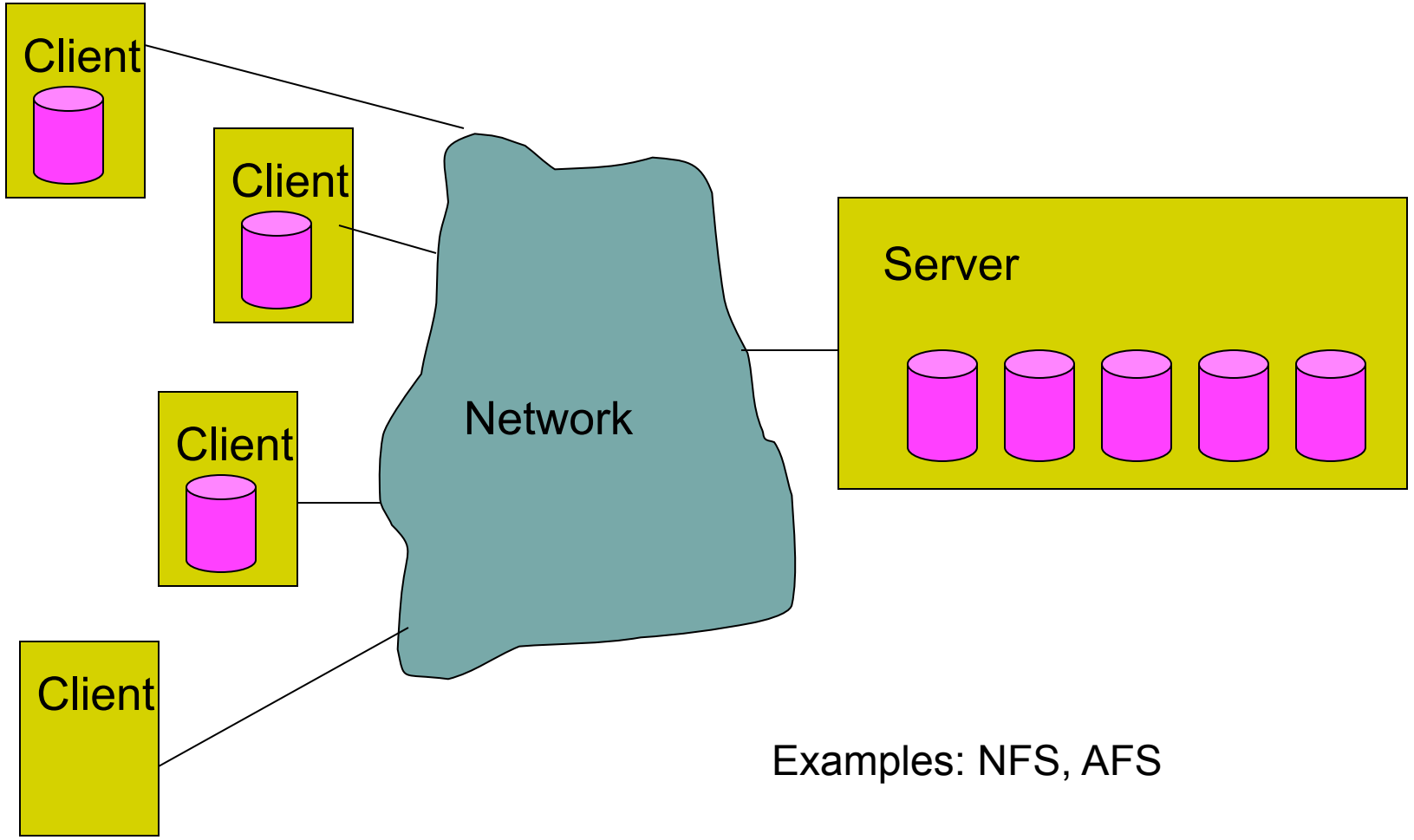
- Plus a number of new issues
  - Distributed agreement (e.g. leader election)
  - Distributed event ordering (e.g. newsgroup)
  - (Partial) Failure recovery

To learn more:



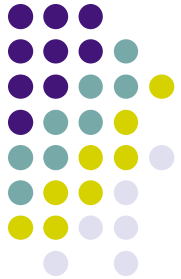
- Join *WukLab*
  - Gain undergraduate research experience
  - System design, building, analysis
  - Focus on datacenter systems (OS, Distributed Systems, Networking, Security, Big Data, etc.)

What is a distributed file system?



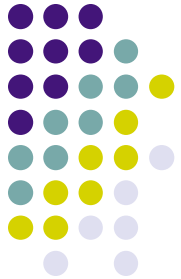
Examples: NFS, AFS

Client/Server Model



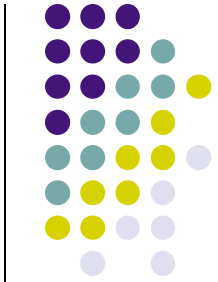
- **Service** – software entity running on one or more machines and providing a particular type of function to a priori unknown clients
- **Server** – which runs the service software to provide the service
- **Client** – process that can invoke a service using a set of operations that forms its *client interface*
- Traditional DFS uses client/server model

Distributed File Systems



- You can login to any instructional machine on campus, your home dir is always there!
- but sometimes it is very slow…

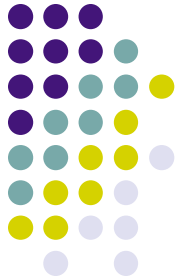
DFS



- Definition: a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources

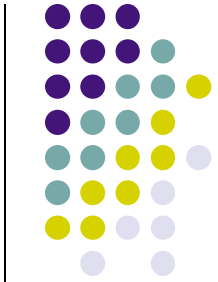
- Many DFS have been proposed and developed

Motivation



- Why are distributed file systems useful?
  - Access from multiple clients
    - Same user on different machines can access same files
  - Simplifies sharing
    - Different users on different machines can read/write to same files
  - Simplifies administration
    - One shared server to maintain (and backup)
  - Improve reliability
    - Add RAID storage to server

Challenges



- Transparent access
  - User sees single, global file system regardless of location
- Scalable performance
  - Performance does not degrade as more clients are added
- Fault Tolerance
  - Client and server identify and respond appropriately when other crashes
- Consistency
  - See same directory and file contents on different clients at same time
- Security
  - Secure communication and user authentication
- Tension across these goals
  - Example: Caching helps performance, but hurts consistency