Role of OS for I/O

- Standard library
  - Provide abstractions, consistent interface
  - Simplify access to hardware devices
- Resource coordination
  - Provide protection across users/processes
  - Provide fair and efficient performance
    - Requires understanding of underlying device characteristics
- User processes do not have direct access to devices
  - Could crash entire system
  - Could read/write data without appropriate permissions
  - Could hog device unfairly
- OS exports higher-level functions
  - File system: Provides file and directory abstractions
  - File system operations: mkdir, create, read, write

File System: OS's storage (I/O) manager

- The concept of a file system is simple
  - the implementation of the abstraction for secondary storage
    - abstraction = files
  - logical organization of files into directories
    - the directory hierarchy
  - sharing of data between processes, people and machines
    - access control, consistency, …

Abstraction: File

- User view
  - Named collection of bytes
    - Untyped or typed
    - Examples: text, source, object, executables, application-specific
  - Permanently and conveniently available
- Operating system view
  - Map bytes as collection of blocks on physical non-volatile storage device
    - Magnetic disks, tapes, flash, NVM
    - Persistent across reboots and power failures
- File system performs the magic / translation
  - Pack bytes into disk blocks on writing
  - Unpack them again on reading

Per-file Metadata

- In Unix, the data representing a file is called an inode (for indirect node)
  - Inodes contain file size, access times, owner, permissions
  - Inodes contain information on how to find the file data (locations on disk)
- Every inode has a location on disk.

System Calls to UNIX File Systems

- 19 system calls into 6 categories:

Return file desp.	Assign inodes	Set file attr.	Process input/ output	Change file system	Modify view of file system
open close creat pipe dup	creat link unlink	chown chmod stat fstat	read write lseek	mount umount	chdir chroot

Steps to Create a File

- 1. Check if name is in use
  - a. Find directory inode
  - b. Read directory contents for existing files
- 2. Allocate inode
  - a. Update from free inode bitmap/list
  - b. Fill in inode contents
- 3. Add an inode to directory
  - a. Write back directory contents
- What happens if you do this in the wrong order?

UNIX Access Rights

- Mode of access: read, write, execute
- Three classes of users

			RWX
a) **owner access**	7	⇒	1 1 1
			RWX
b) **group access**	6	⇒	1 1 0
			RWX
c) **public access**	1	⇒	0 0 1

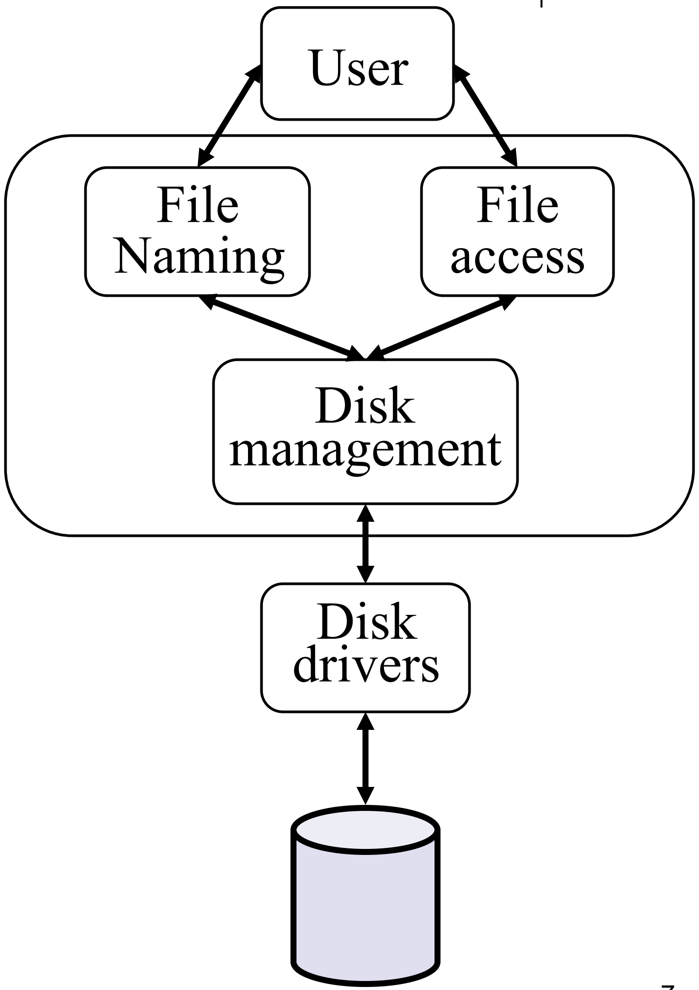
- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.

owner group public

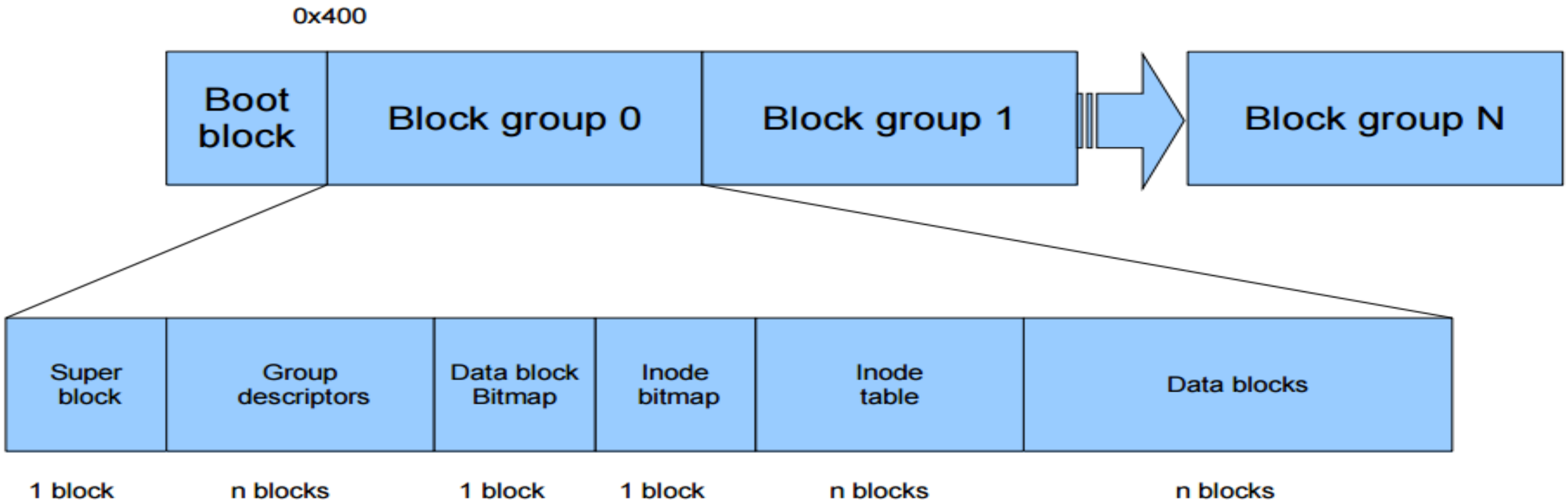
chmod 761 game

File System Components

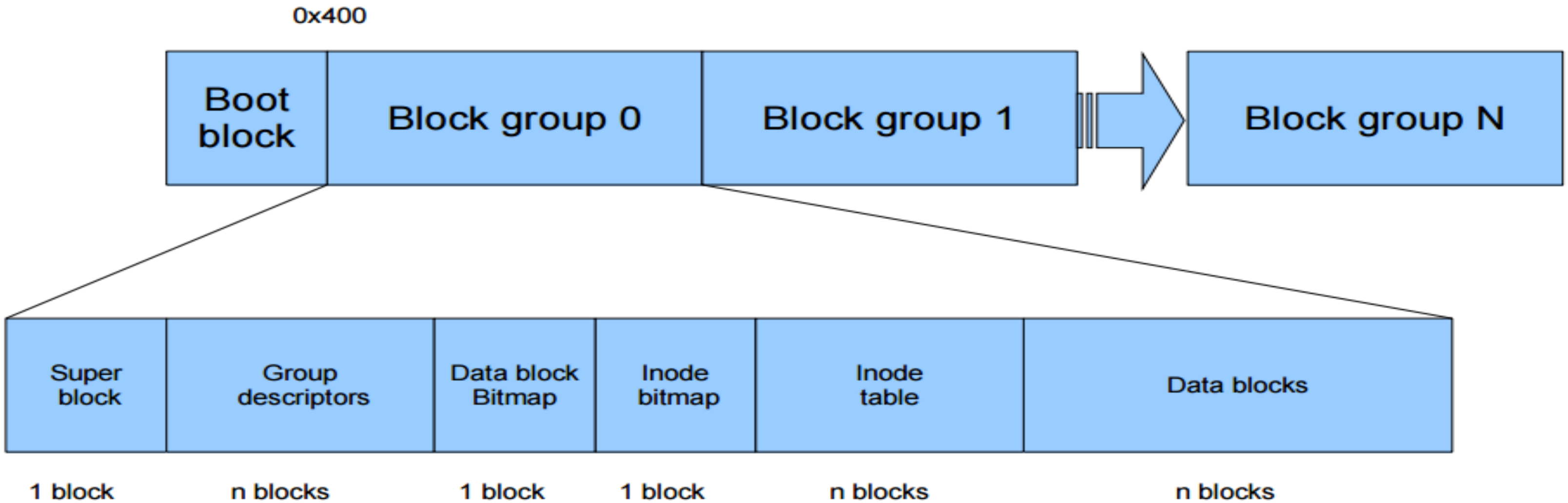
- Naming/Access
  - User gives file name, not track or sector number, to locate data
- Disk management
  - Arrange collection of disk blocks into files
- Protection and permission
  - Protect data from different users
- Reliability/durability
  - When system crashes, lose stuff in memory, but want files to be durable



Disk Layout for a Typical FS



- Boot block: contains info to boot OS
- Block group: next lecture
- Superblock defines a file system
  - type and size of file system
  - size of the file descriptor area
  - free list pointer, or pointer to bitmap
  - location of the file descriptor of the root directory
  - other meta-data such as permission



- What if the superblock is corrupted?
  - What can we do?
  - For reliability, replicate the superblock (in each block group)
- An inode for each file (a header that points to root of the file data blocks) => Inode Table
- Blocks numbered in cylinder-major order, why? (called LBA)
- Data structures to represent free space on disk for both inode and data blocks
  - Bit map: 1 bit per block (sector)
    - How much space does a bit map need for a 4GB disk?
  - Linked list
  - Others?

Data Allocation Problem

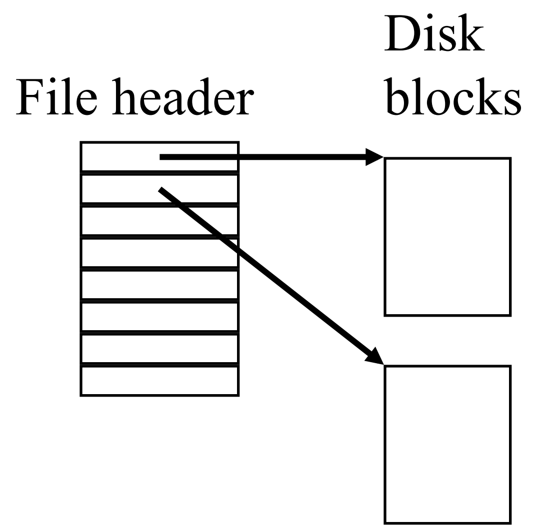
- Definition: allocation data blocks (on disk) when a file is created or grows, and free them when a file is removed or shrinks
- Does this sound familiar?
- Shall we approach it like segmentation or paging?
- What kind of locality matters?
  - Compared to page replacement (on demand paging)?

Hints

- OS allocates LBAs (logical block addresses) to meta-data, file data, and directory data
  - Workload items accessed together should be close in LBA space
- Implications
  - Large files should be allocated sequentially
  - Files in same directory should be allocated near each other
  - Data should be allocated near its meta-data
- Meta-Data: Where is it stored on disk?
  - Embedded within each directory entry
  - In data structure separate from directory entry
    - Directory entry points to meta-data

Indexed Allocation

- Allocate fixed-sized blocks for each file
  - Meta-data: Fixed-sized array of block pointers
    - Allocate space for ptrs at file creation time



Advantages

- No external fragmentation
- Files can be easily grown, with no limit
- Supports random access

Disadvantages

- Large overhead for meta-data:
  - Wastes space for unneeded pointers (most files are small!)

Summary

- Seeks kill performance → exploit spatial locality

- Extent-based allocation optimizes sequential access
- Single-level indexed allocation has speed
- Multi-level index has great flexibility

- Bitmaps show contiguous free space
- Linked lists easy to search for free blocks

Original UFS Problems

- Original Unix FS had two major problems:
  - 1. data blocks are allocated randomly in aging file systems (using linked list)
    - blocks for the same file allocated sequentially when FS is new
    - as FS “ages” and fills, need to allocate blocks freed up when other files are deleted
      - problem: deleted files are essentially randomly placed
      - so, blocks for new files become scattered across the disk!
  - 2. inodes are allocated far from blocks
    - all inodes at beginning of disk, far from data
    - traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks
  - BOTH of these generate many long seeks!

THE FAST FILE SYSTEM（FFS）

- BSD 4.2 introduced the “fast file system”
  - ***Superblock*** is replicated on different cylinders of disk
  - Have one inode table per group of cylinders
    - It minimizes disk arm motions
  - Inode has now 15 block addresses
  - Minimum block size is 4K

Log-Structured File Systems

- LFS was designed in response to two trends in workload and disk technology:
  1. Disk bandwidth scaling significantly (40% a year)
     - but, latency is not
  2. RAM & caches are bigger
  - So, a lot of reads do not require disk access
  - Most disk accesses are writes $\Rightarrow$ pre-fetching not very useful
  - Worse, most writes are small $\Rightarrow$ 10 ms overhead for 50 μs (in mem) write
  - Example: to create a new file:
    - inode of directory needs to be written
    - Directory block needs to be written
    - inode for the file has to be written
    - Need to write the file
  - Delaying these writes could hamper consistency
- Solution: LFS to utilizes full disk bandwidth

LFS Basic Idea

- Structure the disk as a sequential log
  - Periodically, all pending writes buffered in memory are collected in a single segment
  - The entire segment is written contiguously at end of the log
- Segment may contain inodes, directory entries, data
  - Start of each segment has a summary
  - If segment around 1 MB, then full disk bandwidth can be utilized

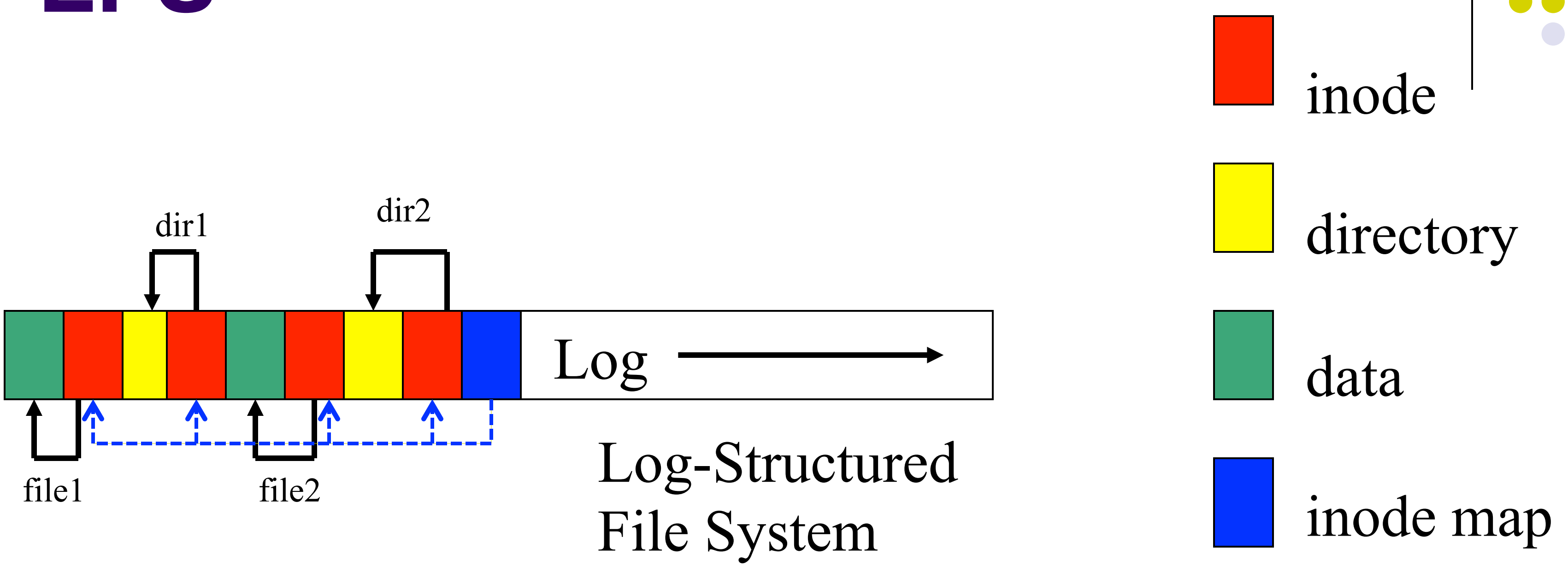
LFS Cleaning

- Finite disk space implies that the disk is eventually full
  - Fortunately, some segments have stale information
  - A file overwrite causes inode to point to new blocks
    - Old ones still occupy space
- Solution: LFS Cleaner thread compacts the log
  - Read segment summary, and see if contents are current
    - File blocks, inodes, etc.
  - If not, the segment is marked free, and cleaner moves forward
  - Else, cleaner writes content into new segment at end of the log
  - The segment is marked as free!
- Disk organized as a circular buffer, writer adds contents to the front, cleaner cleans content from the back

LFS: Locating Data

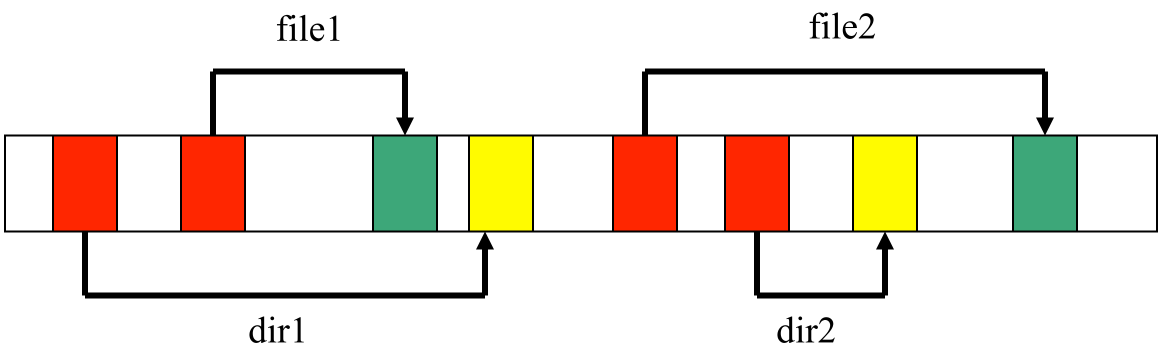
- FFS uses inodes to locate data blocks
  - inodes preallocated in each cylinder group
  - directories contain locations of inodes
- LFS appends inodes to end of log, just like data
  - makes them hard to find
- Solution:
  - use another level of indirection: inode maps
  - inode maps map file #s to inode location
  - location of inode map blocks are kept in a checkpoint region
  - checkpoint region has a fixed location
  - cache inode maps in memory for performance

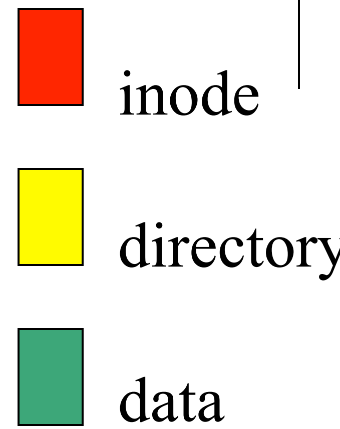
LFS



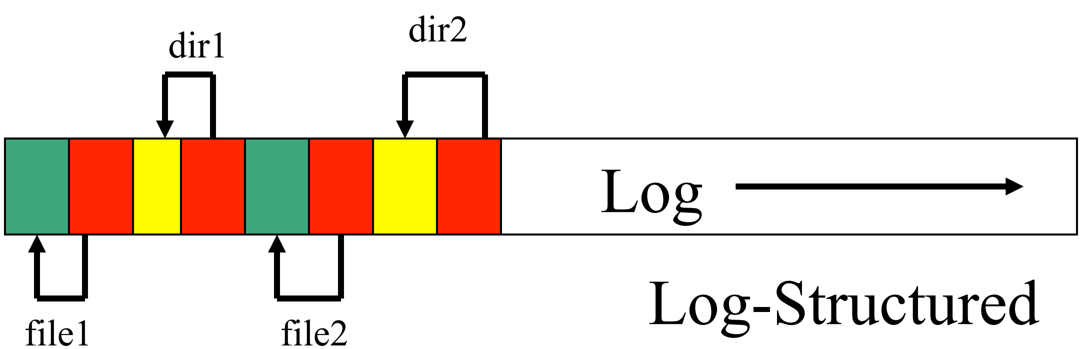
Blocks written to create two 1-block files: dir1/file1 and dir2/file2, in UFS and LFS

UFS vs. LFS





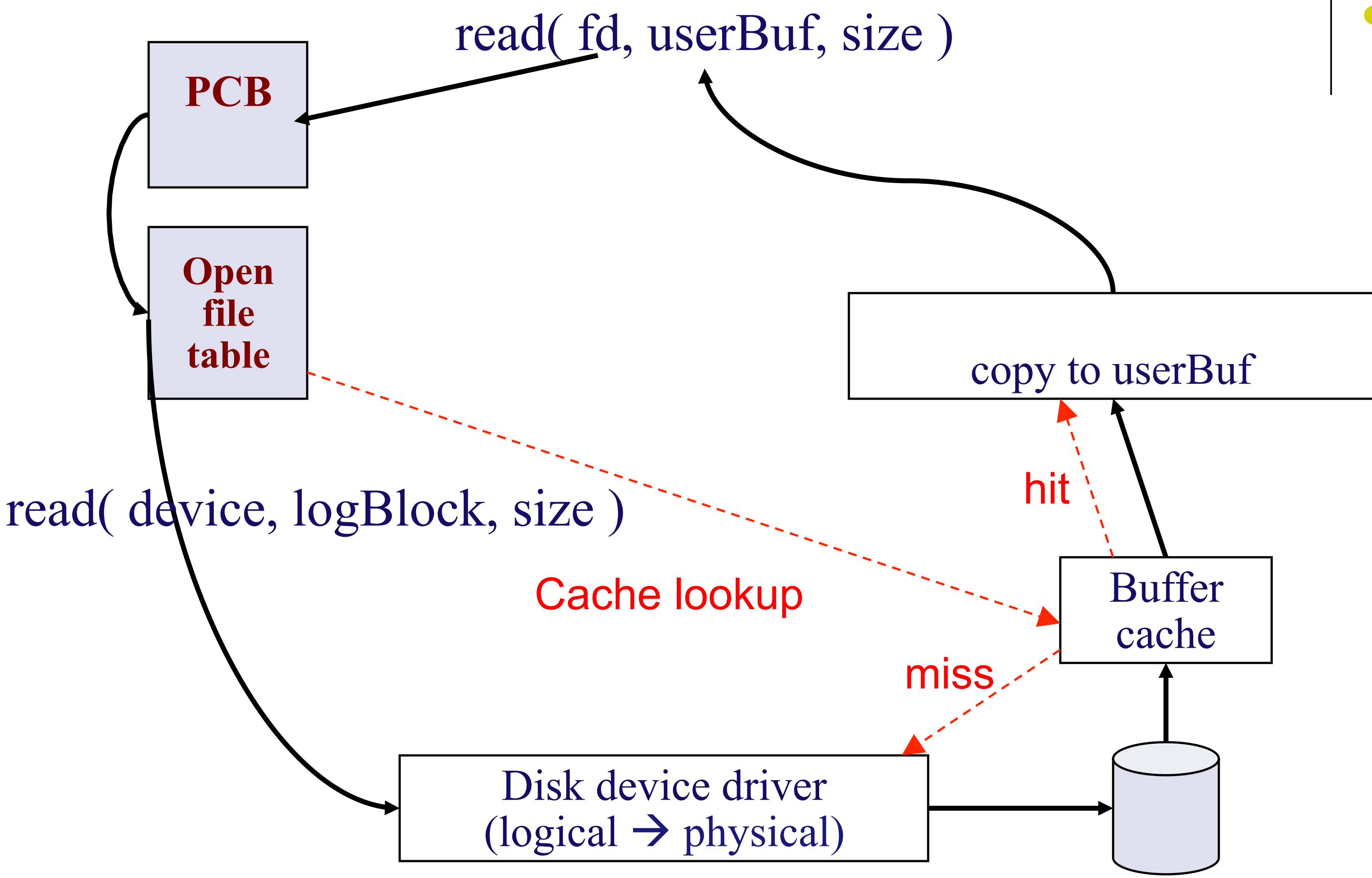
Unix File System



Log-Structured File System

Blocks written to create two 1-block files: dir1/file1 and dir2/file2, in UFS and LFS

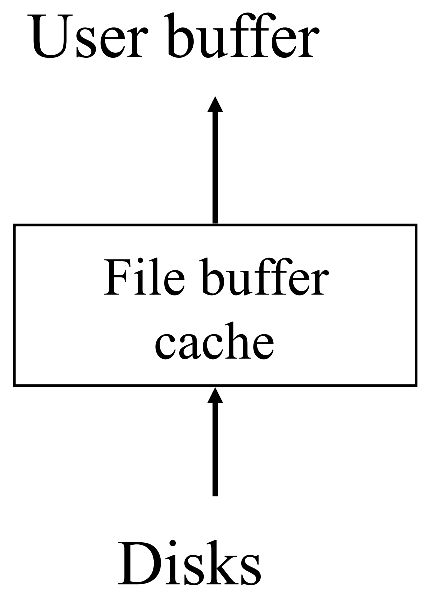
[lec23] Reading A Block



Modern disk drives are addressed as large one-dimensional arrays of logical blocks

Read operations in presence of buffer cache

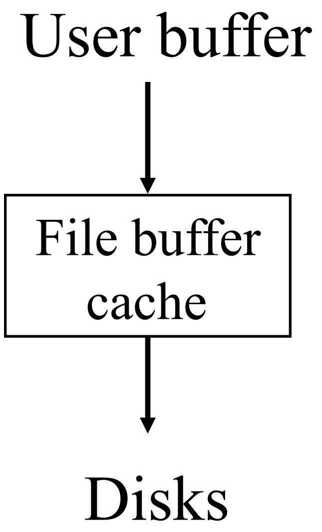
- read(fd, buf, n)
  - On a hit
    - copy from the buffer cache to a user buffer
  - On a miss
    - replacement if necessary
    - read a file block into the buffer cache



Write operations: Maintaining Consistency

- write(fd, buffer, n)
  - On a hit
    - write to buffer cache
  - On a miss
    - Read first
    - Then write (hit)

?



File persistence under file caching

- Problem: fast cache memory is volatile, but users expect disk files to be persistent
  - In the event of a system crash, dirty blocks in the buffer cache are lost !
  - Example 1: creating "/dir/a"
    - Allocate inode (from free inode list) for "a"
    - Update parent dir content – add ("a", inode#) to "dir"
- Solution 1: use write-through cache
  - Modifications are written to disk immediately
    - (minimize "window of opportunities")
  - No performance advantage for disk writes

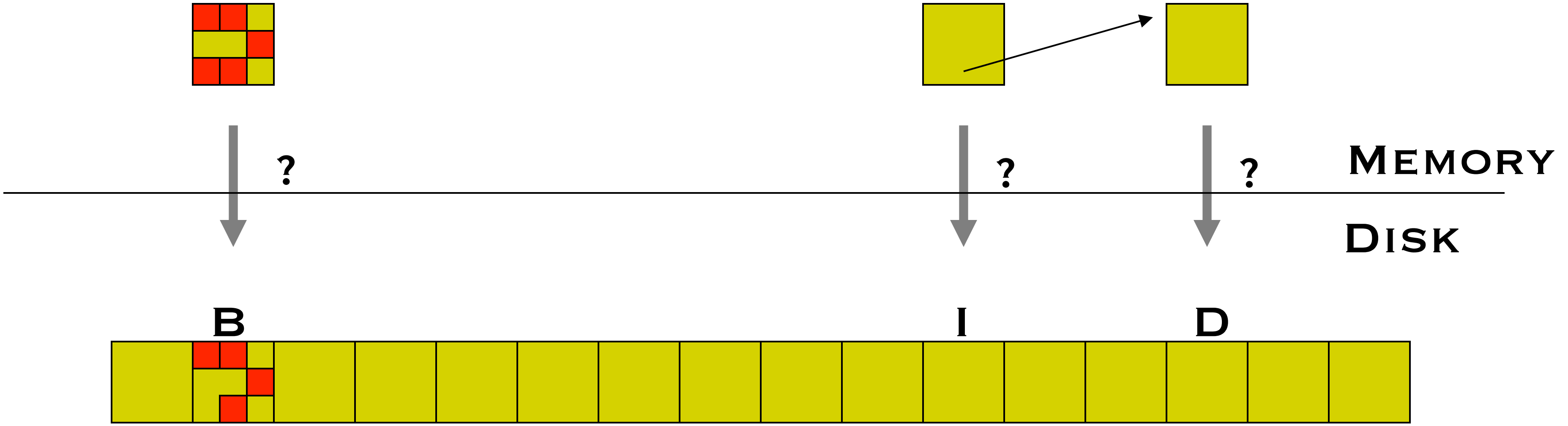
File persistence under file caching

- Possible solution 2: write back cache
  - Gather (buffer) writes in memory and then write all buffered data back to storage devices
  - e.g., write back dirty blocks after no more than 30 seconds
  - e.g., write back all dirty blocks during file close

- Problem with this?

Many "dirty" blocks in memory: What order to write to disk?

- Example: Appending a new block to existing file
  - Write data bitmap B (for new data block),
    write inode I of file (to add new pointer, update time),
    write new data block D



The Problem

- Writes: Have to update disk with N writes
  - Disk does only a single write atomically
- Crashes: System may crash at arbitrary point
  - Bad case: In the middle of an update sequence
- Desire: To update on-disk structures atomically
  - Either all should happen or none

Traditional Solution: FSCK

- FSCK: “file system checker”
- When system boots:
  - Make multiple passes over file system, looking for inconsistencies
    - e.g., inode pointers and bitmaps, directory entries and inode reference counts
  - Either fix automatically or punt to admin
  - Does fsck have to run upon every reboot?
- Main problem with fsck: Performance
  - Sometimes takes hours to run on large disk volumes

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

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

2. Data Replication

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

3. Crash Recovery

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

1. 主动删除 却误删
2. disk failure 硬件
3. FS failure

How to ensure data consistency with arbitrary crash points?

- We need to ensure a “copy” of consistent state can always be recovered

- Either the old consistent state (before udpates)
- Undo Log
  - Make a copy of the old state to a different place
  - Update the current place

- Or the new consistent state (after updates)
- Redo Log
  - Write to a new place, leave the old place intact

The idea of Redo Log

- Idea: Write something down to disk at a different location from the data location
  - Called the "write ahead log" or "journal"
- When all data is written to redo log, write it back to the data location, and then delete the data on redo log

- When crash occurs, look through the redo log and see what was going on
  - Replay complete data, discard incomplete data
  - The process is called "recovery"

Journaling file systems

- Became popular ~2002, but date to early 80's
- There are several options that differ in their details
  - Ntfs (Windows), Ext3 (Linux), ReiserFS (Linux), XFS (Irix), JFS (Solaris)
- Basic idea
  - update metadata, or all data, *transactionally*
    - *"all or nothing"*
    - *Failure atomicity*
  - if a crash occurs, you may lose a bit of work, but the disk will be in a consistent state
    - more precisely, you will be able to quickly get it to a consistent state by using the transaction log/journal – rather than scanning every disk block and checking sanity conditions

Where is the Data?

- In file systems with memory cache, the data is in two places:
  - On disk
  - In in-memory caches
- The basic idea of the solution:
  - Always leave "home copy" of data in a consistent state
  - Make updates persistent by writing them to a sequential (chronological) journal partition/file
  - At your leisure, push the updates (in order) to the home copies and reclaim the journal space
  - Or, make sure log is written before updates

Journal

- Journal: 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 (operations must be idempotent and undoable)
  - <commit t>
    - transaction t has committed – updates will survive a crash
    - Only after the commit block is written is the transaction final
    - The commit block is a single block of data on the disk
- Committing involves writing the records – the home data doesn't need to be updated at this time

How does data get out of the journal?

- After a commit the new data is in the journal – it needs to be written back to its home location on the disk
- Cannot reclaim that journal space until we resync the data to disk

Journal checkpointing

- A cleaner thread walks the journal in order, updating the home locations (on disk, not the cache!) of updates in each transaction
- Once a transaction has been reflected to the home locations, it can be deleted from the journal

How does this help crash recovery?

- Only completed updates have been committed
  - During reboot, the recovery mechanism reapplies the committed transactions in the journal
- The old and updated data are each stored separately, until the commit block is written

If a crash occurs

- Open the log and parse
  - <start>, data, <commit> => committed transactions
  - <start>, no <commit> => uncommitted transactions
- Redo committed transactions
  - Re-execute updates from all committed transactions
  - Aside: note that update (write) is *idempotent*: can be done any positive number of times with the same result.
- Undo uncommitted transactions
  - Undo updates from all uncommitted transactions
  - Write “compensating log records” to avoid work in case we crash during the undo phase

Case Study: Linux ext3

- Ext3: roughly ext2+journaling

- Ext3 grew out of ext2
- Exact same code base
- Completely backwards compatible *(if you have a clean reboot)*

ext3 and journaling

- Two separate layers
  - /fs/ext3 – just the filesystem with transactions
  - /fs/jdb – just the journaling stuff
- ext3 calls jbd as needed
  - Start/stop transaction
  - Ask for a journal recovery after unclean reboot

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

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

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?)

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

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!

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?

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?

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)

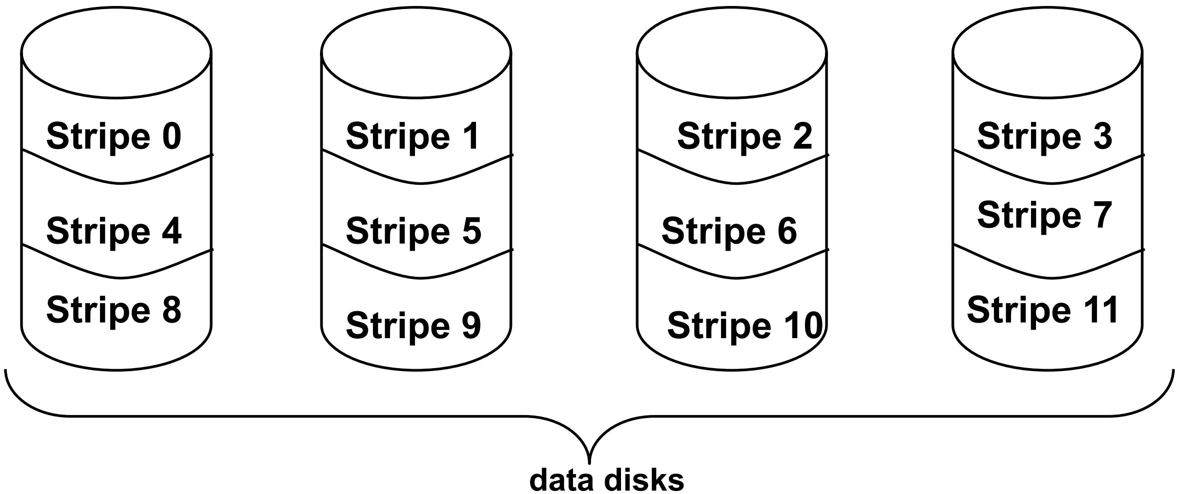
- Mirroring or shadowing (for reliability)
  - Local disk consists of 2 physical disks in parallel
  - Every write performed on both disks
    - Can read from either disk
    - Probability of both fail at the same time?

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?

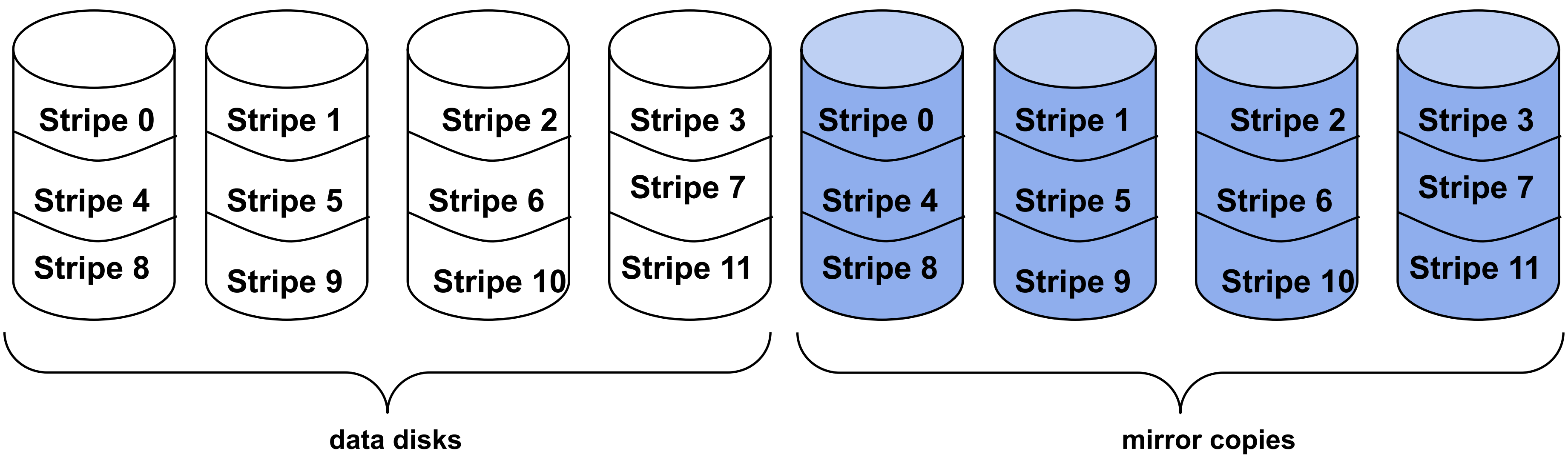
Raid Level 0

- 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?



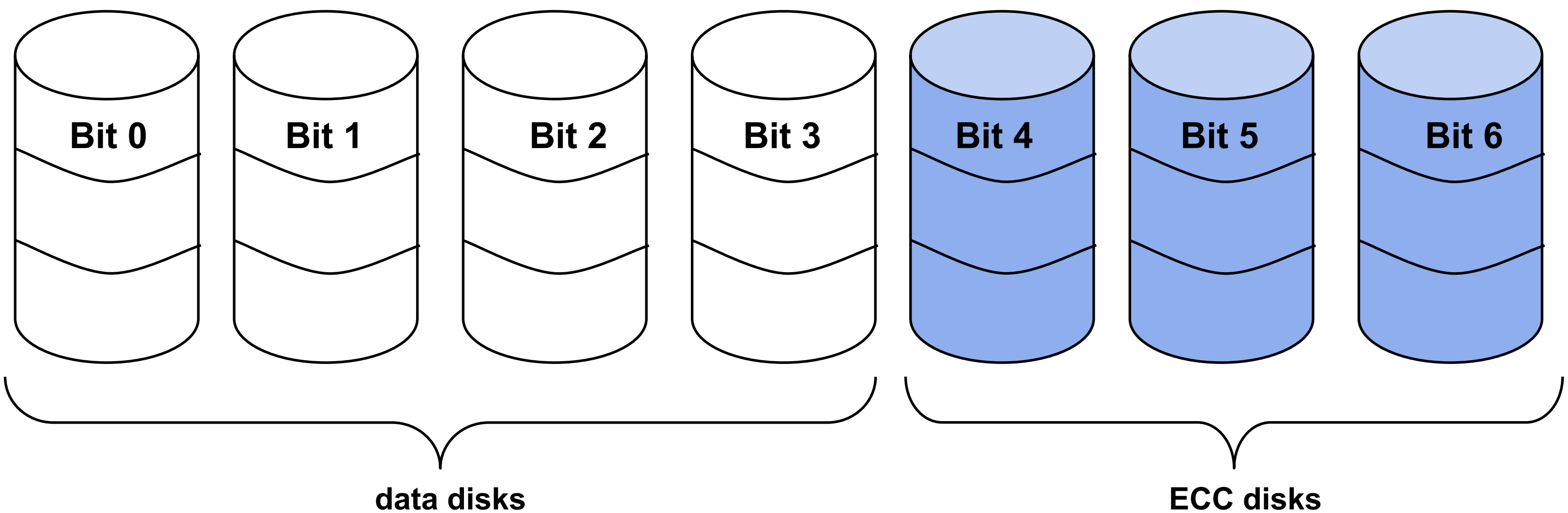
Raid Level 1

- 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



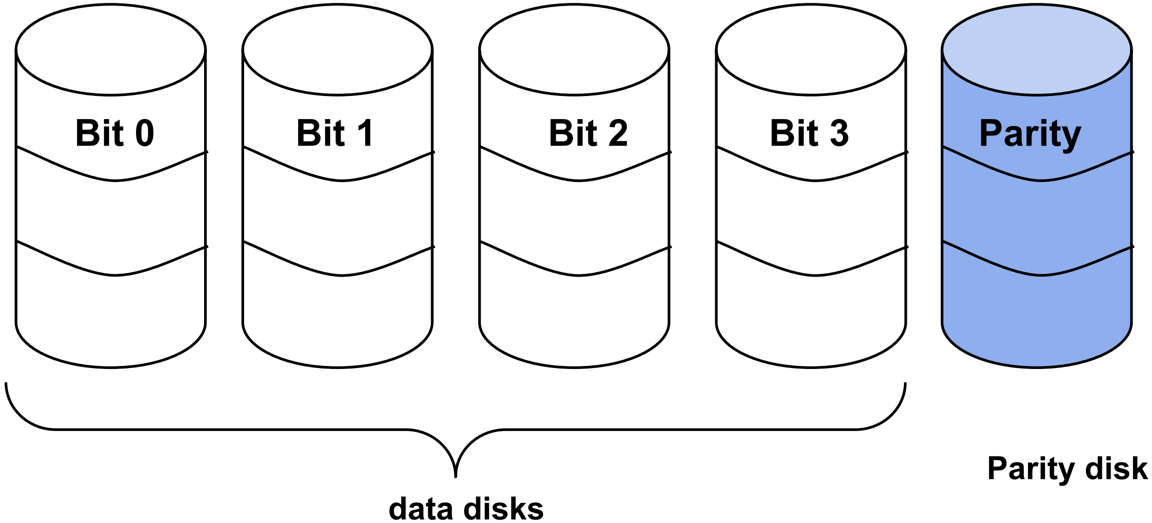
Raid Level 2

- 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

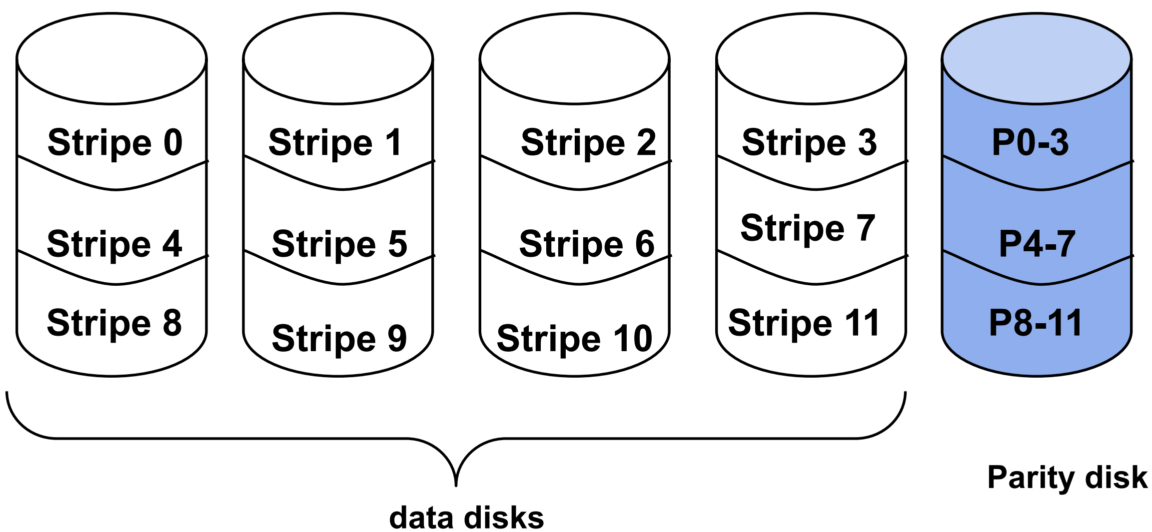
- Use a parity disk
  - Each bit on the parity disk is a parity function of the corresponding bits on all the other disks
- A read accesses all the data disks
- A write accesses all data disks <u>plus</u> the parity disk
- On disk failure, read remaining disks plus parity disk to compute the missing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



Single parity disk can be used to detect and recover errors

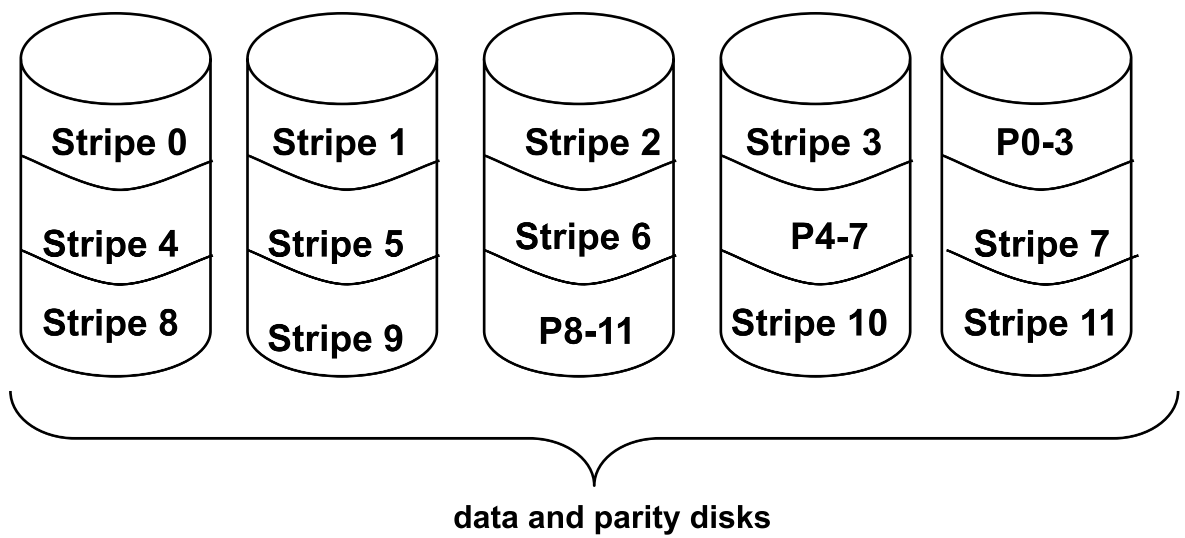
Raid Level 4

- 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



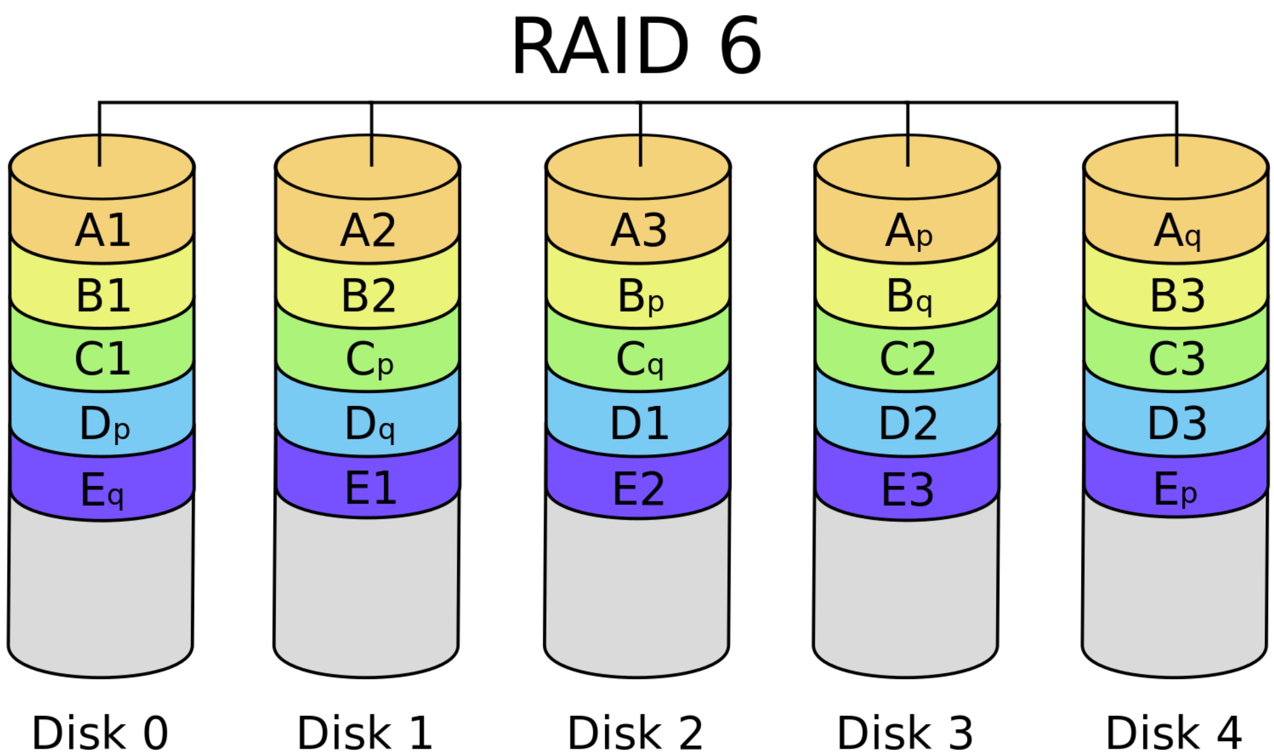
Raid Level 5

- 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

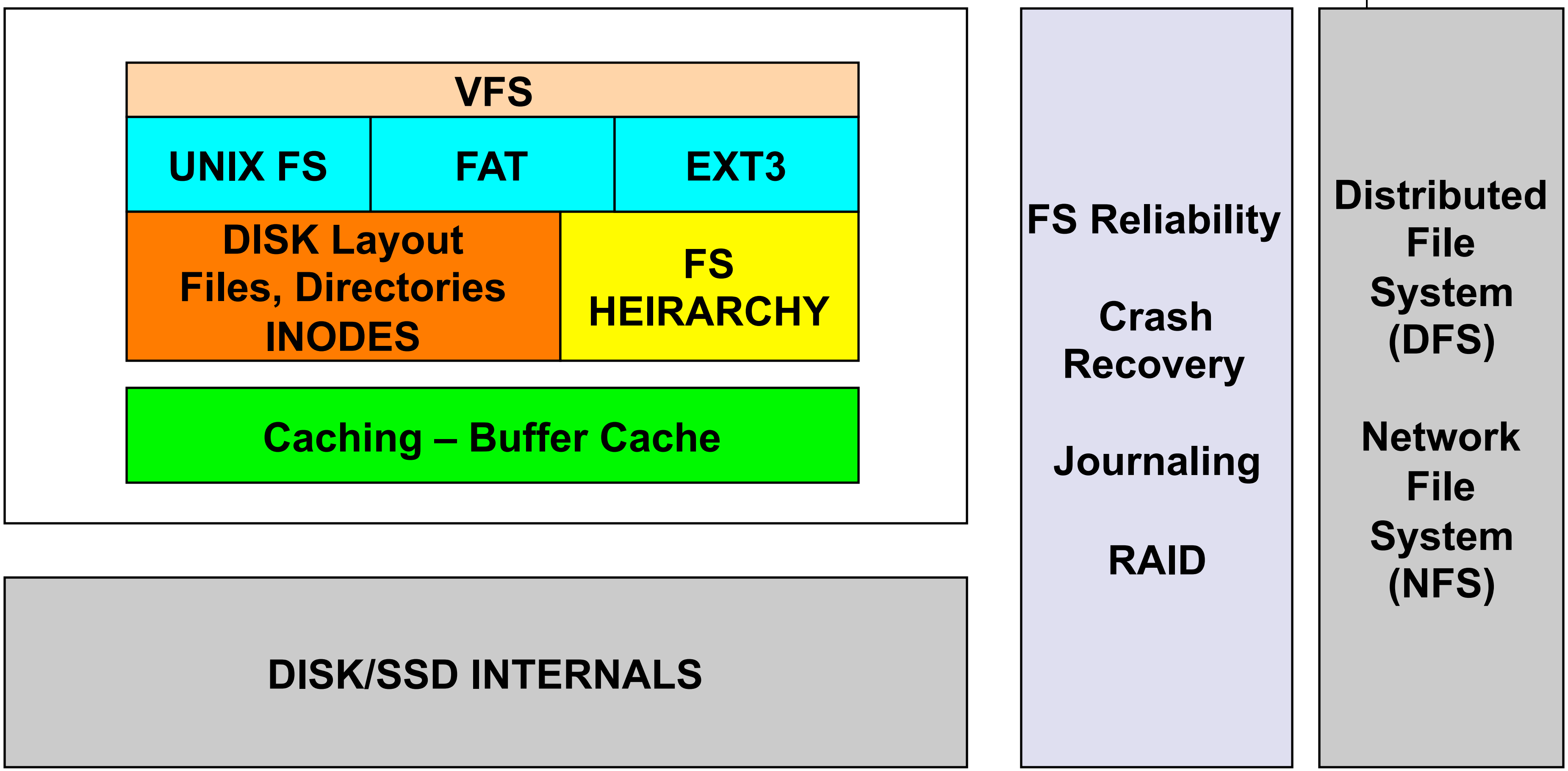


RAID Implementation

- 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

FS topics we have covered



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

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

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

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

NFS Overview

- Remote Procedure Calls (RPC) for communication between client and server
- Client Implementation
  - Provides transparent access to NFS file system
    - UNIX contains Virtual File system layer (VFS)
    - Vnode: interface for procedures on an individual file
  - Translates vnode operations to NFS RPCs
- Server Implementation
  - Stateless: Must not have anything only in memory
  - Implication: All modified data written to stable storage before return control to client
    - Servers often add NVRAM to improve performance

Naming properties

- *Location transparency:*
  - Name of the file does not reveal the file's physical storage location
- *Location independence:*
  - Name of file does not need to be changed when file's physical location changes

NFS Design Objectives

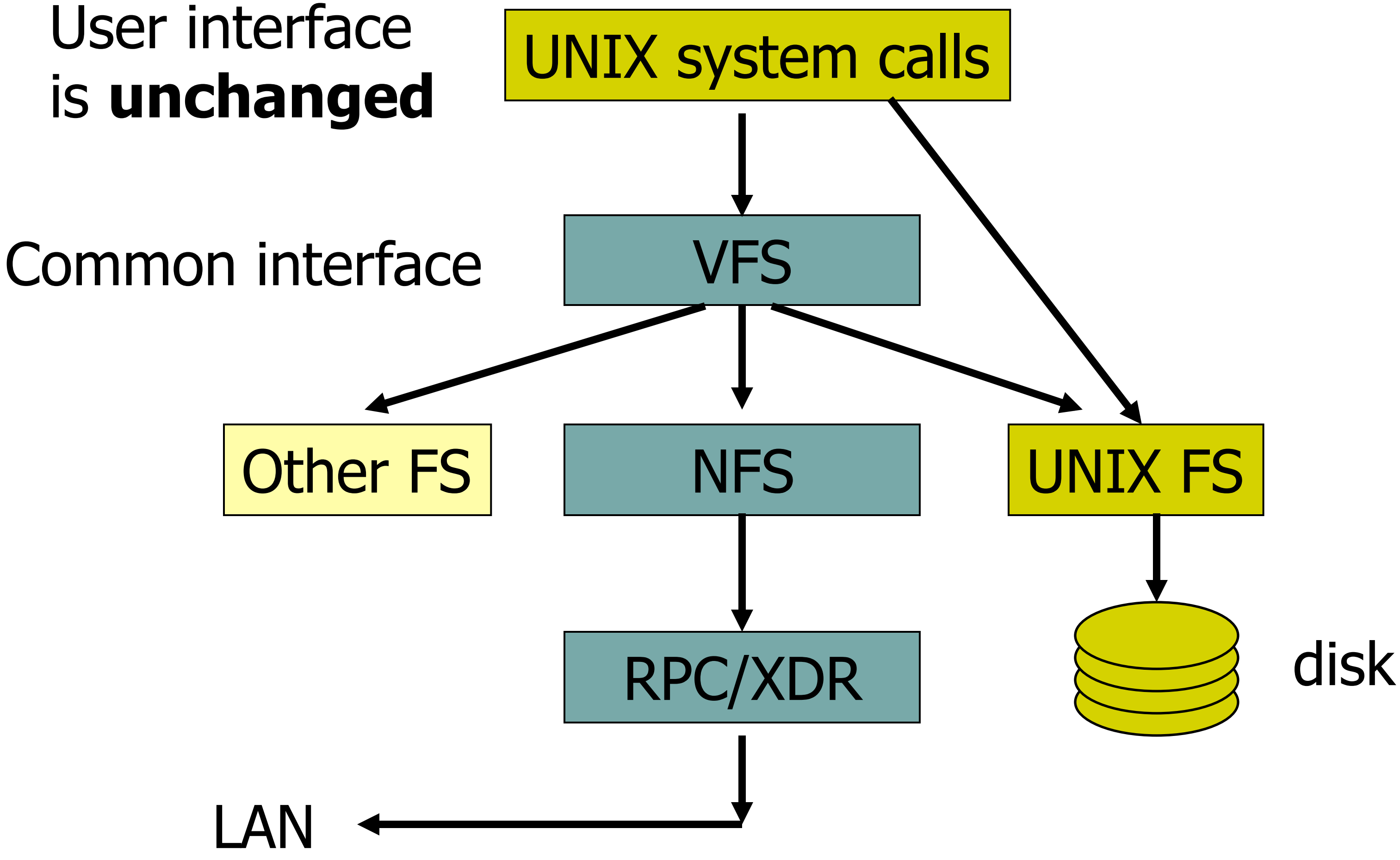
- ***Machine and Operating System Independence***
  - Could be implemented on low-end machines of the mid-80’s
- ***Transparent Access***
  - Remote files should be accessed in exactly the same way as local files
- ***Fast Crash Recovery***
  - Major reason behind stateless design
- ***“Reasonable” performance***
  - Robustness and preservation of UNIX semantics were much more important

Two naming schemes

- Files named by combination of their host name and local name; guarantees a unique systemwide name
  - Neither location transparent
  - Nor location independent

- “Attach” remote directories to local directories, giving the appearance of a coherent directory tree, e.g. Sun’s NFS
- Use internal NFS operations to implement application APIs (POSIX)

Client Side



User interface
is **unchanged**
UNIX system calls
Common interface
VFS
Other FS
NFS
UNIX FS
RPC/XDR
disk
LAN

NFS Design Objectives

- ***Machine and Operating System Independence***
  - Could be implemented on low-end machines of the mid-80's
- ***Transparent Access***
  - Remote files should be accessed in exactly the same way as local files
- ***Fast Crash Recovery***
  - Major reason behind stateless design
- ***"Reasonable" performance***
  - Robustness and preservation of UNIX semantics were much more important

NSF Key Ideas

- NSF key idea #1: Stateless server
  - Server not required to remember anything (in memory)
    - Which clients are connected, which files are open, …
  - Implication: All client requests have all the information to complete op
    - Example: Client specifies offset in file to write to
  - Why is this important for fast crash recovery?
- NSF Key idea #2: Idempotent server operations
  - Operation can be repeated with same result (no side effects)
  - Example: idempotent: a=b+1; Not idempotent: a=a+1;
  - Why is this important for crash recovery?

Advantages of stateless

- Crash recovery is very easy:
  - When a server crashes, client just resends request until it gets an answer from the rebooted server
  - Client cannot tell difference between a server that has crashed and recovered and a slow server
- Server state does not grow with more clients
- Simplifies the protocol
  - Client can always repeat any request

Consequences of stateless

- read and writes calls must specify offset
  - Server does not keep track of current position in the file
- But user will still use conventional UNIX APIs
- How should UNIX APIs be translated?
  - open() / close()
  - read() / write()

How to identify files in NFS?

- Can we still use inode?

- NFS use File handles

- **File handle** consists of
  - ***Filesystem id*** identifying disk partition
  - ***i-node number*** identifying file within partition
  - ***i-node generation number*** changed every time i-node is reused to store a new file

Filesystem id	i-node number	i-node generation number

Basic NFS Protocol

- Operations at NFS layer (applications do not execute these)
  - `lookup(dirfh, name) returns (fh, attributes)`
    - `Use mount protocol for root directory`
  - `create(dirfh, name, attr) returns (newfs, attr)`
  - `remove(dirfs, name) returns (status)`
  - `read(fh, offset, count) returns (attr, data)`
  - `write(fh, offset, count, data) returns attr`
  - `gettattr(fh) returns attr`

Remote lookup

- Returns a **file handle** instead of a file desc.
  - File handle specifies *unique location* of file

- **lookup(dirfh, name)** *returns* **(fh, attr)**
  - Returns file handle **fh** and attributes of named file in directory **dirfh**
  - Fails if client has no right to access directory **dirfh**

Remote lookup

- To lookup **“/usr/joe/6360/list.txt”**

lookup(rootfh, “usr”) returns (fh0, attr)
lookup(fh0, “joe”) returns (fh1, attr)
lookup(fh1, “6360”) returns (fh2, attr)
lookup(fh2, “list.txt”) returns (fh, attr)

Mapping UNIX System Calls to NFS Operations

- Unix system call: `fd = open("/dir/foo")`
  - Traverse pathname to get filehandle for foo
    - `dir_fh = lookup(root_dir_fh, "dir");`
    - `fh = lookup(dir_fh, "foo");`
  - Record mapping from `fd` file descriptor to `fh` NFS filehandle
  - Set initial file offset to 0 for `fd`
  - Return `fd` file descriptor
- Unix system call: `read(fd,buffer,bytes)`
  - Get current file `offset` for `fd`
  - Map `fd` to `fh` NFS filehandle
  - Call `data = read(fh, offset, bytes)` and copy `data` into `buffer`
  - Increment file `offset` by `bytes`
- Unix system call: `close(fd)`
  - Free resources assocatiated with fd
  - No need to tell server: stateless server

NFS Design Objectives

- ***Machine and Operating System Independence***
  - Could be implemented on low-end machines of the mid-80’s
- ***Transparent Access***
  - Remote files should be accessed in exactly the same way as local files
- ***Fast Crash Recovery***
  - Major reason behind stateless design
- ***“Reasonable” performance***
  - Robustness and preservation of UNIX semantics were much more important

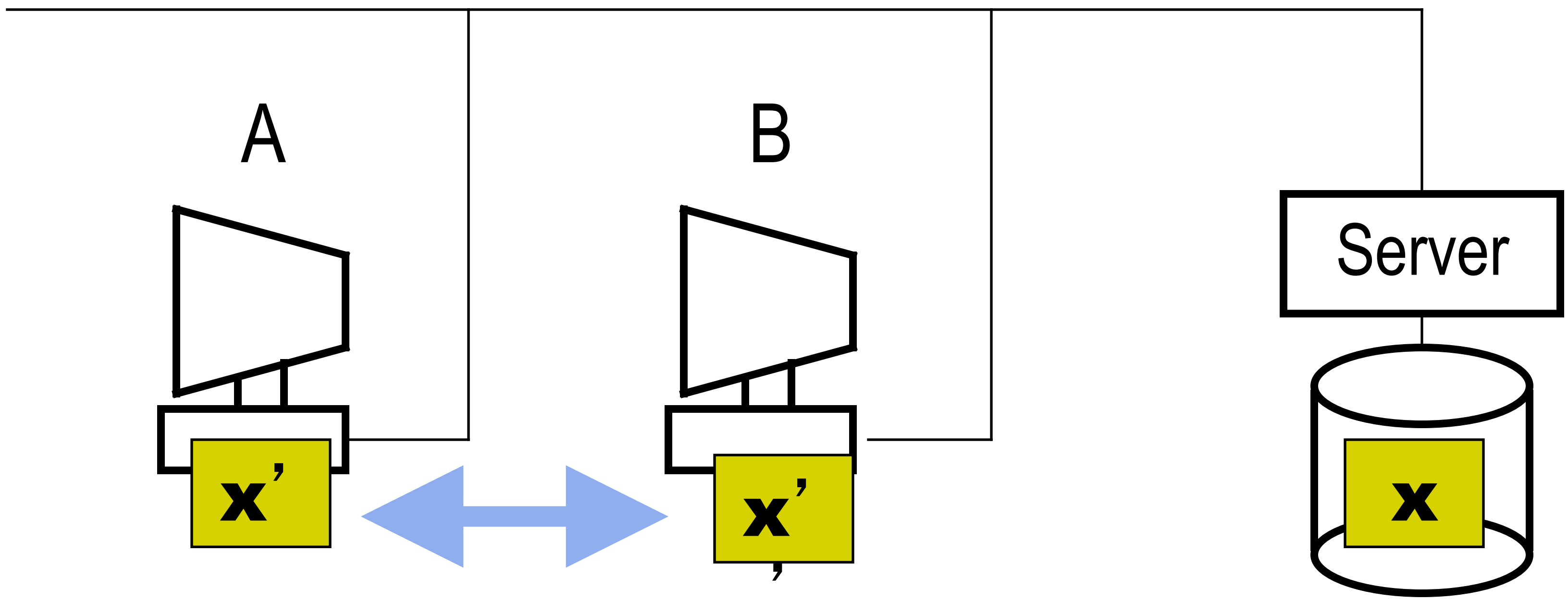
Client-side Caching

- Caching needed to improve performance
  - Reads: Check local cache before going to server
  - Writes: Only periodically write-back data to server
  - Why avoid contacting server
    - Avoid slow communication over network
    - Server becomes scalability bottleneck with more clients
- Two types of client caches
  - data blocks
  - attributes (metadata)

Cache Consistency

- Problem: Consistency across multiple copies (server and multiple clients)
  - How to keep data consistent between client and server?
    - If file is changed on server, will client see update?
    - Determining factor: Read policy on clients
  - How to keep data consistent across clients?
    - If write file on client A and read on client B, will B see update?
    - Determining factor: Write and read policy on clients

Cache Consistency Problem



Inconsistent updates

NFS Consistency: Reads

- Reads: How does client keep current with server state?
  - Attribute cache: Used to determine when file changes
    - File open: Client checks server to see if attributes have changed
      - If haven't checked in past T seconds (configurable, T=3)
    - Discard entries every N seconds (configurable, N=60)
  - Data cache
    - Discard all blocks of file if attributes show file has been modified
- Eg: Client cache has file A's attributes and blocks 1, 2, 3
  - Client opens A:
  - Client reads block 1 => cache
  - Client waits 70 seconds
  - Client reads block 2 => cache
  - Block 3 is changed on server
  - Client reads block 3 => cache, get old value
  - Client reads block 4 => fetch from server
  - Client waits 70 seconds
  - Client reads block 1 => fetch from server

NFS Consistency: Writes

- Writes: How does client update server?
  - Files
    - Write-back from client cache to server every 30 seconds
    - Also, Flush (write all dirty data) on close() (AKA *flush-on-close*)
  - Directories
    - Synchronously write to server (write through)

- Example: Client X and Y have file A (blocks 1,2,3) cached
  - Clients X and Y open file A
  - Client X writes to blocks 1 and 2
  - Client Y reads block 1 => cache
  - 30 seconds later...
  - Client Y reads block 2 => cache, get old value
  - 40 seconds later...
  - Client Y reads block 1 => server

Conclusions

- Distributed file systems
  - Important for data sharing
  - Challenges: Fault tolerance, scalable performance, and consistency

- NFS: Popular distributed file system
  - Key features:
    - Stateless server, idempotent operations: Simplifies fault tolerance
    - Crashed server appears as slow server to clients
  - Client caches needed for scalable performance
    - Rules for invalidating cache entries and flushing data to server are not straight-forward
    - Data consistency very hard to reason about

Course review – 2nd half semester

Page Replacement Policies

- Optimal
- Random
- FIFO
  - Belady's anomaly
- Approximate LRU, NRU
- FIFO with 2nd chance
- Clock: a simple FIFO with 2nd chance

More Virtual Memory

- Thrashing
- Working set model, size, replacement aglorithm

- Shared memory
- COW

Virtual Memory Questions to Think About

- What is the use of optimal algo?
- If future is unknown, what make us think there is a chance for doing a good job?
- Without addi. hardware support, the best we can do?
- What is the minimal hardware support under which we can do a decent job?
- Why is it difficult to implement exact LRU? Exact anything
- For a fixed replacement algo, more page frames → less page faults?
- How can we move page-out out of critical path?

More Virtual Memory Questions

- Per-process vs. global page replacement
- Thrashing
- What causes thrashing?
- What to do about thrashing?
- What is working set?
- What’s the benefit of Copy-on-Write?

Practice Question (Quiz 2)

- Consider the following virtual page reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
- How many page faults will there be under the LRU replacement algorithm on Nick's PC which has 4 physical pages? How many page faults will there be on Riley's machine which has 3 physical pages?

- Nick's: 8
- Riley's: 10

Storage and File System

Storage Device

- Disk Internals
  - Seek/rotation, random/sequential accesses
- SSD Internals
  - Flash read/write/erase, the granularity of them
  - Erase-before-write, flash wear
  - FTLs
  - Garbage collection and wear leveling
- Both Disks and SSDs can fail!

RAID

- Two motivations
  - Performance, reliability
- Two main ideas
  - Striping, mirroring (parity)

no RAID level 2 3 in exam

- RAID levels: 1-6

Practice question

- Assume that
  - you have a mixed configuration comprising disks organized as RAID Level 1 and as RAID Level 5 disks;
  - the system has flexibility in deciding which disk organization to use for storing a particular file.
  - you have a mixed workload of *frequently-read* and *frequently-written* files
- Which files should be stored in the RAID Level 1 disks and which in the RAID Level 5 disks in order to optimize performance?

File System Overview

- File system abstraction
  - File, directory, FS APIs

- Directories
  - Different directory organizations
  - Directory internals
  - Path walk
  - Hard link, soft link

- Metadata
  - Inode

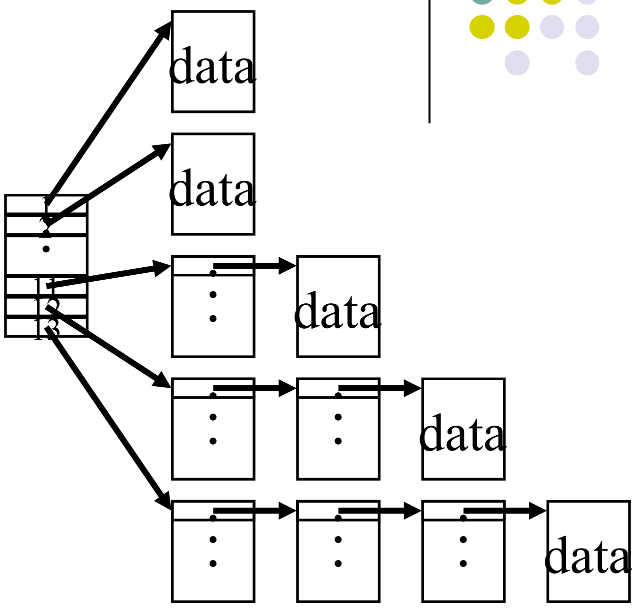
File Allocation

- Two tasks:
  - How to allocate blocks for a file?
  - How to design inode to keep track of blocks?
- Allocation methods:
  - Contiguous
  - Extent-based
  - Linked
  - File-allocation Tables
  - Indexed
  - Multi-level Indexed
- Free space management
  - linked list
  - bitmap

UFS

- UFS
  - multi-level indexed files
  - Inodes all stored on outermost track
  - Free-block linked list
  - Two problems of UFS
    - data blocks are allocated randomly in aging file systems
    - inodes are allocated far from blocks

Indirect blocks addressing ranges



- Assume blocksize = 1K
  - a block contains 1024 / 4 = 256 block addresses
- direct block address: 10K
  - indirect block addresses: 256K
  - double indirect block addresses: 256 * 256K = 64M
  - tripe indirect block addresses: 256 * 64M = 16G

What happens in accessing block 23, 5, 340?

FFS and LFS

- FFS
  - Cylinder group
    - Inodes and data within same dir
  - Free block managed with bitmap
- File system buffer cache
- LFS
  - All writes buffered into chunks and go to an append-only log
  - Locating data is more difficult
  - Needs background cleaner

Journaling File System

- Data reliability
  - Three threats: three directions to prevent data loss
- The problem with write back cache under system crash
  - One file system operation consists of multiple sub-operations, they should be atomic
- Journaling
  - Write to a redo log first, in a transactional way
  - Journal checkpointing, crash recovery
  - Ext3: three journaling modes

Distributed File System

- DFS and client/server model

- NFS
  - Transparency through indirection of VFS
  - Two key ideas for fast crash recovery
    - Stateless server
    - Idempotent operations
  - Client cache and cache consistency

Great ideas in Computer System Design (1)

- “*All computer science problems can be solved with an extra level of indirection*”

  -- David Wheeler

1. Dynamic memory relocation
   - Base&bound, segmentation, paging
2. One-level paging → Two-level paging
3. UFS multi-level indexed files
4. NFS: transparency via VFS

Great ideas in Computer System Design (2)

- Principle of locality → Caching

1. TLB
2. Demand paging (VM)
3. Buffer cache in FS
4. (On-disk cache)
5. (Client caching in NFS)

5. (Hardware cache, L1, L2, etc.)