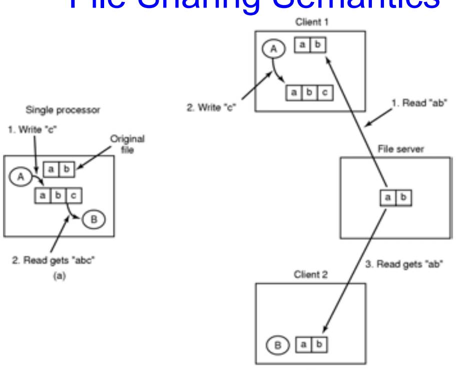
Other File Systems: NFS, FFS, WAFL, LFS

Distributed File Systems

- Goal: view a distributed system as a file system
  - Storage is distributed
  - Web tries to make world a collection of hyperlinked documents
- Issues not common to usual file systems
  - Naming transparency
  - Load balancing
  - Scalability
  - Location and network transparency
  - Fault tolerance
- We will look at some of these today

File Sharing Semantics



Caching

- Keep repeatedly accessed blocks in cache
  - Improves performance of further accesses
- How it works:
  - If needed block not in cache, it is fetched and cached
  - Accesses performed on local copy
  - One master file copy on server, other copies distributed in DFS
  - Cache consistency problem: how to keep cached copy consistent with master file copy
- Where to cache?
  - Disk: Pros: more reliable, data present locally on recovery
  - Memory: Pros: diskless workstations, quicker data access,
  - Servers maintain cache in memory

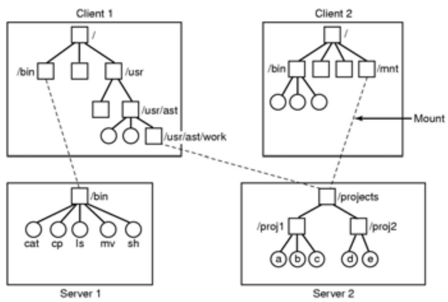
File Sharing Semantics

- Other approaches:
  - Write through caches:
    - immediately propagate changes in cache files to server
    - Reliable but poor performance
  - Delayed write:
    - Writes are not propagated immediately, probably on file close
    - Session semantics: write file back on close
    - Alternative: scan cache periodically and flush modified blocks
    - Better performance but poor reliability
  - File Locking:
    - The upload/download model locks a downloaded file
    - Other processes wait for file lock to be released

Network File System (NFS)

- Developed by Sun Microsystems in 1984
  - Used to join FSes on multiple computers as one logical whole
- Used commonly today with UNIX systems
- Assumptions
  - Allows arbitrary collection of users to share a file system
  - Clients and servers might be on different LANs
  - Machines can be clients and servers at the same time
- Architecture:
  - A server exports one or more of its directories to remote clients
  - Clients access exported directories by mounting them
    - The contents are then accessed as if they were local

Example



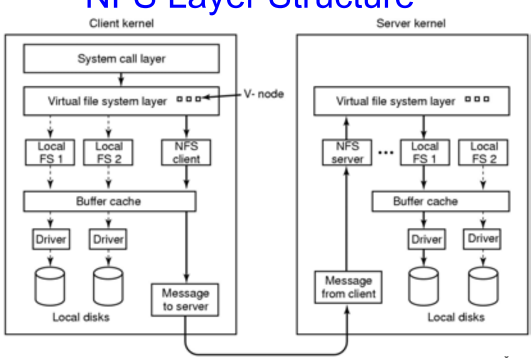
7

NFS Protocol

- Supports directory and file access via RPCs
- All UNIX system calls supported other than *open* & *close*
- *Open* and *close* are intentionally not supported
  - For a *read*, client sends *lookup* message to server
  - Server looks up file and returns handle
  - Unlike *open*, *lookup* does not copy info in internal system tables
  - Subsequently, *read* contains file handle, offset and num bytes
  - Each message is self-contained
- Pros: server is stateless, i.e. no state about open files
- Cons: Locking is difficult, no concurrency control

8

NFS Layer Structure



-

Summary of Issues for File Systems

- Creating/representing/destroying independent files.
  - disk block allocation, file block map structures
  - directories and symbolic naming
- Masking the high seek/rotational latency of disk access.
  - smart block allocation on disk
  - block caching, read-ahead (prefetching), and write-behind
- Reliability and the handling of updates.

Note: This and remaining slides stolen from Jeff Chase, Duke University

10

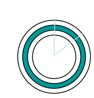
The Problem of Disk Layout

- The level of indirection in the file block maps allows flexibility in file layout.
  - "File system design is 99% block allocation." [McVoy]
- Competing goals for block allocation:
  - *allocation cost*
  - *bandwidth* for high-volume transfers
  - *disk utilization*
  - *efficient directory operations*
- **Goal:** reduce disk arm movement and seek overhead.
  - metric of merit: *bandwidth utilization* (or *effective bandwidth*)

11

FFS Cylinder Groups

- FFS defines *cylinder groups* as the unit of disk locality, and it factors locality into allocation choices.
  - typical: thousands of cylinders, dozens of groups
  - Strategy: place "related" data blocks in the same cylinder group whenever possible.
    - seek latency is proportional to seek distance
  - Smear large files across groups:
    - Place a run of contiguous blocks in each group.
  - Reserve inode blocks in each cylinder group.
    - This allows inodes to be allocated close to their directory entries and close to their data blocks (for small files).



12

FFS Allocation Policies

- Allocate file inodes close to their containing directories.
  - For *mkdir*, select a cylinder group with a more-than-average number of free inodes.
  - For *creat*, place inode in the same group as the parent.
- Concentrate related file data blocks in cylinder groups.
  - Most files are read and written sequentially.
  - Place initial blocks of a file in the same group as its inode.
    - How should we handle directory blocks?
  - Place adjacent logical blocks in the same cylinder group.
    - Logical block ***n+1*** goes in the same group as block ***n.***
    - Switch to a different group for each indirect block.

The Problem of Metadata Updates

- Metadata updates are a second source of FFS seek overhead.
  - Metadata writes are poorly localized.
    - E.g., extending a file requires writes to the inode, direct and indirect blocks, cylinder group bit maps and summaries, and the file block itself.
- Metadata writes can be delayed, but this incurs a higher risk of file system corruption in a crash.
  - If you lose your metadata, you are dead in the water.
  - FFS schedules metadata block writes carefully to limit the kinds of inconsistencies that can occur.
    - Some metadata updates must be synchronous on controllers that don't respect order of writes.

FFS Failure Recovery

- FFS uses a two-pronged approach to handling failures:
- Carefully order metadata updates to ensure that no dangling references can exist on disk after a failure.
  - Never recycle a resource (block or inode) before zeroing all pointers to it (*truncate, unlink, rmdir*).
  - Never point to a structure before it has been initialized.
    - E.g., sync inode on *creat* before filling directory entry, and sync a new block before writing the block map.
- Run a file system *scavenger* (*fsck*) to fix other problems.
  - Free blocks and inodes that are not referenced.
  - Fsck will never encounter a dangling reference or double allocation.

Allocating a Block in FFS

- 1. Try to allocate the rotationally optimal physical block after the previous logical block in the file.
  - Skip *rotdelay* physical blocks between each logical block.
  - (rotdelay is 0 on track-caching disk controllers.)
- 2. If not available, find another block a nearby rotational position in the same cylinder group
  - We'll need a short seek, but we won't wait for the rotation.
  - If not available, pick any other block in the cylinder group.
- 3. If the cylinder group is full, or we're crossing to a new indirect block, go find a new cylinder group.
  - Pick a block at the beginning of a run of free blocks.

Clustering in FFS

- *Clustering* improves bandwidth utilization for large files read and written sequentially.
  - Allocate clumps/clusters/runs of blocks contiguously; read/write the entire clump in one operation with at most one seek.
  - Typical cluster sizes: 32KB to 128KB.
- FFS can allocate contiguous runs of blocks "most of the time" on disks with sufficient free space.
  - This (usually) occurs as a side effect of setting *rotdelay* = 0.
    - Newer versions may relocate to clusters of contiguous storage if the initial allocation did not succeed in placing them well.
  - Must modify buffer cache to group buffers together and read/write in contiguous clusters.

Effect of Clustering

Access time = seek time + rotational delay + transfer time

average seek time = 2 ms for an intra-cylinder group seek, let's say
rotational delay = 8 milliseconds for full rotation at 7200 RPM: average delay = 4 ms
transfer time = 1 millisecond for an 8KB block at 8 MB/s

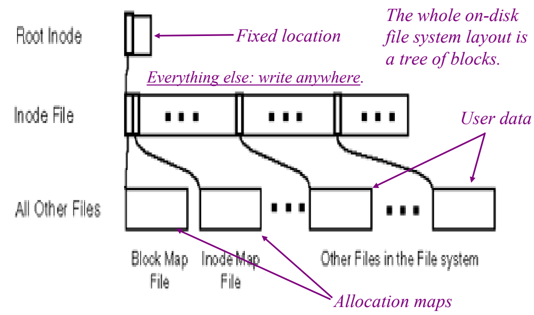
8 KB blocks deliver about 15% of disk bandwidth.
64KB blocks/clusters deliver about 50% of disk bandwidth.
128KB blocks/clusters deliver about 70% of disk bandwidth.

Actual performance will likely be better with good disk layout, since most seek/rotate delays to read the next block/cluster will be "better than average".

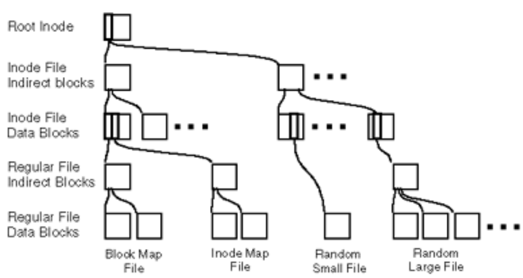
WAFL: Write Anywhere File Layout

- Used in the Network Appliance NFS Server
- Optimizes file system for random reads

WAFL: High-Level View



WAFL: A Closer Look

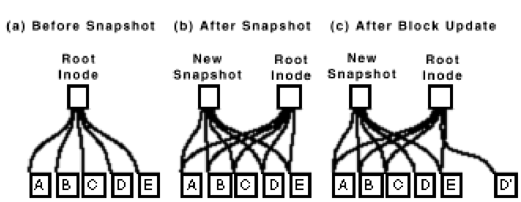


Snapshots

"WAFL's primary distinguishing characteristic is Snapshots, which are readonly copies of the entire file system."

This was really the origin of the idea of a *point-in-time copy* for the file server market. What is this idea good for?

Snapshots



The snapshot mechanism is used for user-accessible snapshots and for transient *consistency points.*

How is this like a *fork*?

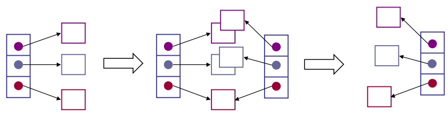
WAFL Consistency Points

- "WAFL uses Snapshots internally so that it can restart quickly even after an unclean system shutdown."
- "A *consistency point* is a completely self-consistent image of the entire file system. When WAFL restarts, it simply reverts to the most recent consistency point."
  - Buffer dirty data in memory (delayed writes) and write new consistency points as an atomic batch (*force*).
  - A consistency point transitions the FS from one self-consistent state to another.
  - Combine with NFS operation log in NVRAM
    - What if NVRAM fails?

Shadowing

Shadowing is the basic technique for doing an atomic *force*.

reminiscent of *copy-on-write*



1. starting point
modify purple/grey blocks

2. write new blocks to disk
prepare new block map

3. overwrite block map
(*atomic commit*)
and free old blocks

Frequent problems: nonsequential disk writes, damages clustered allocation on disk. *How does WAFL deal with this?*

The Nub of WAFL

- WAFL's consistency points allow it to buffer writes and push them out in a batch.
  - Deferred, clustered allocation
  - Batch writes
  - Localize writes
- Indirection through the metadata "tree" allows it to write data wherever convenient: the tree can point anywhere.
  - Maximize the benefits from batching writes in consistency points.
  - Also allow multiple copies of a given piece of metadata, for snapshots.

Alternative: Logging and Journaling

- *Logging* can be used to localize synchronous metadata writes, and reduce the work that must be done on recovery.
    - Universally used in database systems.
    - Used for metadata writes in journaling file systems (e.g., Episode).
- Key idea: group each set of related updates into a single log record that can be written to disk *atomically* ("all-or-nothing").
  - Log records are written to the log file or log disk *sequentially*.
    - No seeks, and preserves temporal ordering.
  - Each log record is trailed by a marker (e.g., checksum) that says "this log record is complete".
  - To recover, scan the log and reapply updates.

Metadata Logging

- Here's one approach to building a fast filesystem:
- 1. Start with FFS with clustering.
- 2. Make all metadata writes asynchronous.
- ***But***, that approach cannot survive a failure, so:
- 3. Add a supplementary log for modified metadata.
- 4. When metadata changes, write new versions immediately to the log, *in addition to* the asynchronous writes to "home".
- 5. If the system crashes, recover by scanning the log.
    - Much faster than scavenging *(fsck)* for large volumes.
- 6. If the system does not crash, then discard the log.

Log-Structured File System (LFS)

- In LFS, *all* block and metadata allocation is log-based.
  - LFS views the disk as "one big log" (logically).
  - *All* writes are clustered and sequential/contiguous.
    - Intermingles metadata and blocks from different files.
  - Data is laid out on disk in the order it is written.
  - No-overwrite allocation policy: if an old block or inode is modified, write it to a new location at the *tail* of the log.
  - LFS uses (mostly) the same metadata structures as FFS; only the allocation scheme is different.
    - Cylinder group structures and free block maps are eliminated.
    - Inodes are found by indirecting through a new map (the *ifile*).

Writing the Log in LFS

- LFS "saves up" dirty blocks and dirty inodes until it has a full *segment* (e.g., 1 MB).
  - Dirty inodes are grouped into block-sized clumps.
  - Dirty blocks are sorted by *(file, logical block number)*.
  - Each log segment includes summary info and a checksum.
- LFS writes each log segment in a single burst, with at most one seek.
  - Find a free segment "slot" on the disk, and write it.
  - Store a back pointer to the previous segment.
    - Logically the log is sequential, but physically it consists of a chain of segments, each large enough to amortize seek overhead.

Cleaning in LFS

- What does LFS do when the disk fills up?
- As the log is written, blocks and inodes written earlier in time are superseded (“killed”) by versions written later.
  - files are overwritten or modified; inodes are updated
  - when files are removed, blocks and inodes are deallocated
- A cleaner daemon compacts remaining live data to free up large hunks of free space suitable for writing segments.
  - look for segments with little remaining live data
  - write remaining live data to the log tail
  - can consume a significant share of bandwidth, and there are lots of cost/benefit heuristics involved.