File Systems

- "A Fast File System for UNIX"
  ACM TOCS 1984, McKusick et. al.
- "The Design and Implementation of a Log-Structured File System"
  SOSP'91 and ACM TOCS, Rosenblum, Ousterhout
- "Improving the Performance of Log-Structured File Systems with Adaptive Methods"
  SOSP'97, Neefe-Matthews et. al.

Fast File System (FFS) for UNIX

- Old file system (before 1984)
  - *inode*: (owner, timestamp of mods and access times, array of indices to data blocks)
  - allocation is suboptimal
    - never transfers more than 512 bytes
    - even sequential data blocks require seeks
    - limited TPUT
  - simple improvements
    - block size increase to 1024 bytes
    - still poor utilization of disk bandwidth
  - indications
    - increase block size
    - reduce seeks (scrambled free list)

FFS Organization

- Min size of file system block: 4096 bytes
- Blocks equal or greater to any power of two greater than 4096 bytes
- Block size is decided when file is created
- Divides disk partition in "cylinder groups"
- Tries to transfer blocks in a single disk transaction
- But what about small files?
- Division of single file system block into "fragments"
- Lower bound of fragment size: 512 bytes
- Allocation is done with "write"
- As good binpacking as possible (but still fragmentation...)
- Free space reserve

File System Parameterization

- Hardware parameters
- Hardware characteristics of disk (rotational times, number of blocks per track, etc)
- Define distance between rotational optimal blocks
- "rotational layout tables"

Layout Policies

- Increase locality (minimize seeks)
- But spread out uncorrelated data
- Two parts
- Global policies
  - inode allocation: for a directory, on the same cylinder
  - data blocks: put them on the same cylinder group, preferably at rotationally optimal positions
- Local policies
  - how to lay out data blocks
  - four-level allocation strategy
  - rotationally closest
  - first on same cylinder, then same cylinder group, then look for another cylinder group (hash), then exhaustive search

Log-Structured File System

- Motivation: technology trends
  - CPU-memory-disk
  - rapid increases in CPU and memory performance
  - disk technology increases: transfer bandwidth (e.g., RAIDs) and capacity
  - but disks bound by mechanical motions
  - big memory: "absorbs" reads,can be used for write-back policies (thus, large sequential transfers)
  - disk traffic dominated by writes
- Workloads: office and engineering are dominated by small files, a lot of time spend on "metadata"
- scientific workloads: different story

Existing file systems...

- Spread info on disk: too many small accesses
- "inode" separate from file's contents, directory entry
- Usually, five disk I/Os each preceded by a seek:
  - two accesses to files attributes
  - one access to file's data
  - one access to directory's data
  - one access to directory's attributes
- Too expensive for small files!
- Writes: synchronous
- Low performance
- Even in FFS, metadata are written synchronously

Log Structured

- Focus: optimize writes
- General idea
  - writes all mods to disk in a log-like sequential structure
  - single disk write operation: utilizes all disk bandwidth
  - eliminates seeks
  - speeds up file writing and fast recovery from crashes
- Need large sequential space on disk
- "segments": components of log
- "segment cleaner": compress live info
- In summary, outperforms Unix for small writes, as good as standard Unix for large reads and writes

- Issues
  - how to retrieve info (file location and reading)
  - how to manage space on disk so large chunks are available
- File location and reading
  - inode is the basic data structure
  - file attributes: (type, owner, position, permissions, disk address of first 10 blocks, possible addresses of indirect blocks)
  - inode positions: inode *map*, not at fixed positions
  - Figure 1

- Space management: segments
  - fragmentation
  - choices
    - threading: keep track of what is left
    - copying: copy and compact space
    - disadvantage: cost
    - combo of threading and copying
    - split disk into segments
    - collect long-lived data into segments – do not touch those!

Segment Cleaning Mechanism

- Copy live data out of a segment: cleaning
- Read segments into memory
- Identify live data
- Write live data back to a smaller number of clean segments
- Use segment summary info to determine policy issues
  - when should we do the cleaning? (background?)
  - how many segments to clean at a time? (the more the better?)
  - which ones? (the most fragmented!)
  - how to group live ones together? (same directory? same age?)

- N segments
- u utilization
- Write Cost = $\frac{total\ bytes\ read\ and\ written}{new\ data\ written}$ = $\frac{read\ segs\ +\ write\ live\ +\ write\ new}{new\ data\ writing} = \frac{N+N*u+N*(1-u)}{N*(1-u)} = \frac{2}{1-u}$
- Figure 3
- Target: bimodal distribution
  - very full segments
  - very empty segments

- Simulation
- Uniform distribution
- Hot and cold files (sorted by age, 10% of files, but selected 90% of the time and 90% of files are selected 10% of the time)
- Results: cleaner must treat hot and cold segments differently
- Hot should not be cleaned more often than cold!
- Benefit is not great (prone to fragmentation)
- Use "aging" for stability (cold)
- New policy: $\frac{benefit}{cost} = \frac{(1-u)*age}{1+u}$
- Cold segments are cleaned at a much higher utilization than hot ones
- Age: segment usage table

- Crash recovery
- Two stages
- Check for inconsistencies
- Check-pointing: keep track of last disk operation
- Roll-forward: consistent to last checkpoint
- Look at inodes and blocks that are written, and also their updates

Advantages/Disadvantages of LFS

- Advantages: good for
  - frequent small writes
  - reads absorbed by cache
  - sufficient time to clean the log
- Disadvantages
  - workloads with random updates and high disk utilization [Seltz93]
  - high cleaning cost
- Suggestions
  - adaptive techniques
  - measuring and self-tuning
  - workload and hardware

"Improving Performance of LFS with Adaptive Methods"

- Optimizations
  - Choose segment size to trade-off transfer efficiency against cleaning efficiency
  - dynamically choose cleaning method
  - factor cache contents to reduce the cost of cleaning
  - reorganize disk contents to improve disk performance

Improving Write Performance

- High performance across a range of workloads
- Initial implementations: all segments have the same size
- If variable sizes, smaller are more likely to empty completely before cleaning
- Competing effects
- Vary segment size!
- Solution: cleaner observes holes and workloads with locality (smaller segments)

Adaptive Cleaning

- Choose garbage collection strategy based on usage patterns
- Avoid cases where the cleaner must process many full segments
- “hole-plugging”: write live blocks in holes of other segments
- High cost, but in cases it is worth it!
- Choose adaptively between traditional cleaning and hole-plugging
- Candidate segments, estimate cost-benefit of each approach, decide

Cached Data

- Reduce write cost
- If a segment is completely cached, can be cleaned!
- Avoids disk reads
- Prefer those segments versus the non-cached ones!

Improve Read Performance

- Read and write patterns do not match!
- Reorganize!
- History of previous events (block access graph)
- Layout for near-optimal performance (minimize mechanical delays)
- Differences of current and desired layout
- May be reorganize (at convenient times)