CSE 451: Operating Systems
Spring 2013

Module 17
Journaling File Systems

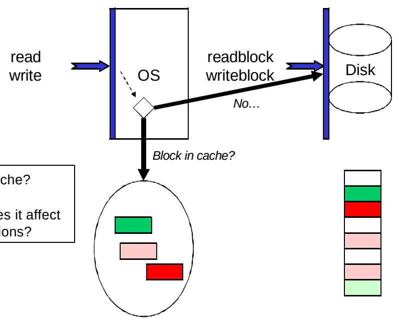
Ed Lazowska
lazowska@cs.washington.edu
Allen Center 570

In our most recent exciting episodes …

- Original Bell Labs UNIX file system
  – a *simple yet practical* design
  – exemplifies engineering tradeoffs that are pervasive in system design
  – elegant but slow
    - and performance gets worse as disks get larger
- BSD UNIX Fast File System (FFS)
  – solves the throughput problem
    - larger blocks
    - cylinder groups
    - aggressive caching
    - awareness of disk performance details

Caching (applies both to FS and FFS)

- Cache (often called *buffer cache*) is just part of system memory
- It's system-wide, shared by all processes
- Need a replacement algorithm
  – LRU usually
- Even a relatively small cache can be very effective
- Today's huge memories => bigger caches => even higher hit ratios
- Many file systems "read-ahead" into the cache, increasing effectiveness even further



- Why cache?
- How does it affect applications?

Caching writes => problems when crashes occur

- Some applications assume data is on disk after a write (seems fair enough!)
- And the file system itself will have (potentially costly!) consistency problems if a crash occurs between syncs – i-nodes and file blocks can get out of whack
- Approaches:
  – "write-through" the buffer cache (synchronous – too slow),
  – NVRAM: write into battery-backed RAM (too expensive) and then later to disk, or
  – "write-behind": maintain queue of uncommitted blocks, periodically flush (unreliable – this is the sync solution – used in FS and FFS)

FS and FFS are real dogs when a crash occurs

- Caching is necessary for performance
- Suppose a crash occurs during a file creation:
  1. Allocate a free inode
  2. Point directory entry at the new inode
- In general, after a crash the disk data structures may be in an inconsistent state
  – metadata updated but data not
  – data updated but metadata not
  – either or both partially updated
- fsck (i-check, d-check) are *very* slow
  – must touch every block
    - Must do this in "file system order" not in "disk order"
      – Disk copy is fast; file copy is slow
  – worse as disks get larger!

Journaling file systems

- Became popular ~2002
- There are several options that differ in their details
  - Ext3, ReiserFS, XFS, JFS, ntfs
- Basic idea
  - update metadata, or all data, *transactionally*
    - *"all or nothing"*
  - 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 the file systems we have seen already, the data is in two places:
  - On disk
  - In in-memory caches
- The caches are crucial to performance, but also the source of the potential "corruption on crash" problem
- 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

Redo log

- Log: an append-only file containing log records
  - <start t>
    - transaction t has begun
  - <t,x,v>
    - transaction t has updated block x and its new value is v
      - Can log block "diffs" instead of full blocks
  - <commit t>
    - transaction t has committed – updates will survive a crash
- Committing involves writing the redo records – the home data needn't be updated at this time

If a crash occurs

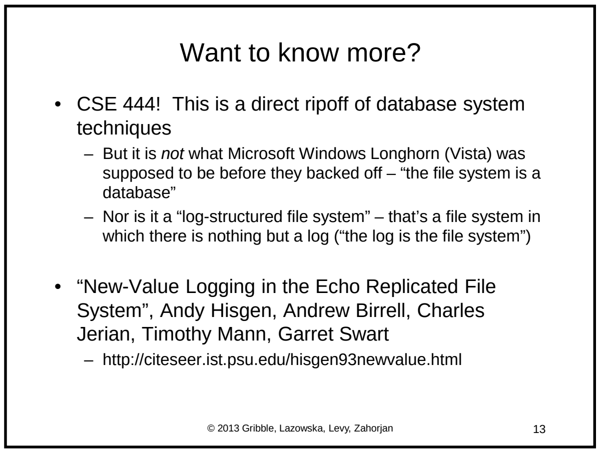
- Recover the log
- Redo committed transactions
  - Walk the log in order and re-execute updates from all committed transactions
  - Aside: note that update (write) is *idempotent*: can be done any non-zero number of times with the same result.
- Uncommitted transactions
  - Ignore them. It's as though the crash occurred a tiny bit earlier…

Managing the Log Space

- A "cleaner" thread walks the log in order, updating the home locations of updates in each transaction
  - Note that idempotence is important here – may crash while cleaning is going on
- Once a transaction has been reflected to the home blocks, it can be deleted from the log

Impact on performance

- The log is a big contiguous write
  - very efficient
- And you do fewer synchronous writes
  - these are very costly in terms of performance
- So journaling file systems can actually improve performance (immensely)
- As well as making recovery very efficient



Want to know more?

- CSE 444! This is a direct ripoff of database system techniques
  - But it is *not* what Microsoft Windows Longhorn (Vista) was supposed to be before they backed off – “the file system is a database”
  - Nor is it a “log-structured file system” – that’s a file system in which there is nothing but a log (“the log is the file system”)

- “New-Value Logging in the Echo Replicated File System”, Andy Hisgen, Andrew Birrell, Charles Jerian, Timothy Mann, Garret Swart
  - http://citeseer.ist.psu.edu/hisgen93newvalue.html