

CSE 451: Operating Systems

Spring 2017

Module 15

Journaling File Systems

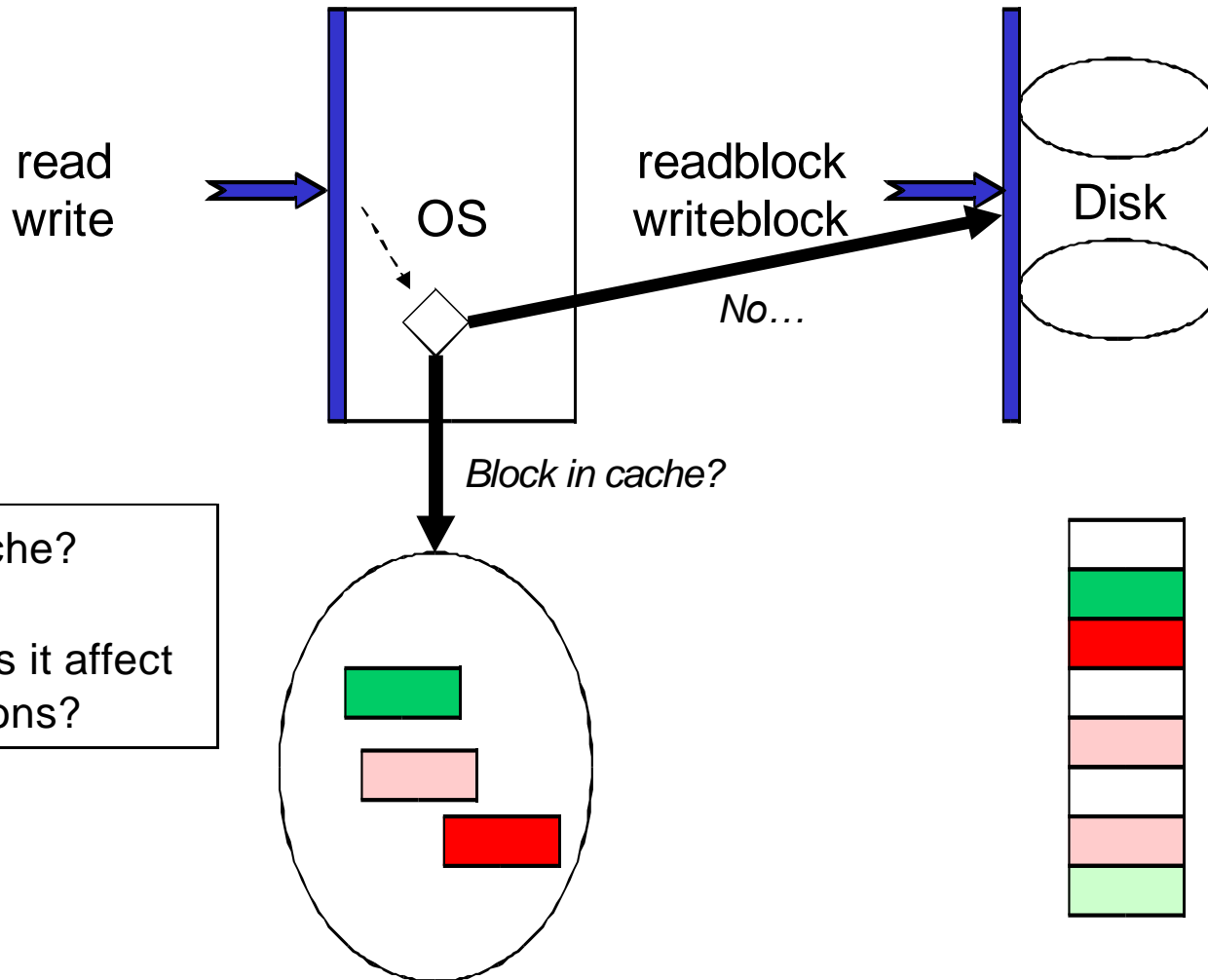
John Zahorjan

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)
 - addresses 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 may have consistency problems if a crash occurs between syncs – i-nodes and file blocks can get out of sync
- Approaches:
 - “write-through” the buffer cache (synchronous – too slow – plus it doesn’t solve the problem!),
 - 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
- Imagine creating a new file
 - Have to allocate an i-node (write i-node map)
 - Have to initialize new i-node (write i-node)
 - Have to create a directory entry (write directory i-node, directory data block, and data map if had to allocate new block for directory)
 - Have to update superblock (free data and i-node counts)
- Suppose a crash occurs while that's going on...

Anticipating crashes

- You have to choose an order to perform writes
- What order works?
 - Have to allocate an i-node (write i-node map)
 - Have to initialize new i-node (write i-node)
 - Have to create a directory entry (write directory i-node, directory data block, and data map if had to allocate new block for directory)
 - Have to update superblock (free data and i-node counts)
- fsck (i-check, d-check) are *very* slow
 - must touch nearly every block
 - Must do this in “file system order” not in “disk order”
 - Disk copy is fast; file copy is slow
 - gets 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
- Goal: Make sure on-disk data is always in a consistent state
- How?
 - update metadata [and 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
- In a journaling file system, the data may be in three places:
 - The cache
 - The “home copy” on disk
 - A journal entry on disk
- The journal contains updates to the home copy blocks

What about performance?

- Have to do two writes for each update
 - one for journal entry and one to home location
 - that can't be good...
- Most reads/writes are absorbed by the cache
 - You must eventually write, though
 - Imagine a burst of file creation
- The journal can **help** performance
 - write big segments of journal entries sequentially on the disk
 - (each entry indicates the new value of some disk block)
 - sequential writes are much faster than random writes
 - At your leisure, push the updates (in order) to the home copies and reclaim the journal space

Redo log

- Log: a chronologically ordered, append-only file containing log records
 - $\langle \text{start } t \rangle$
 - transaction t has begun
 - $\langle t, x, v \rangle$
 - transaction t has updated block x and its new value is v
 - log block “diffs” instead of full blocks
 - $\langle \text{commit } t \rangle$
 - transaction t has *committed*
- A transaction whose commit record makes it into the on-disk journal survives a crash
- A transaction whose commit record doesn't make it will be discarded

If a crash occurs

- Re-execute the log's operations
- 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...
 - Sure, you lose some work (updates), but the file system isn't corrupted

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>

So... what can go wrong?

- Sometimes the log gets corrupted
 - Data errors
 - Controller/firmware errors
- What do you do now?

Hoo boy...

- Checking NTFS involves
 - One seek to read an MFT entry
 - One seek to verify metadata match with directory
 - And... reading and verifying the btree
- That's $\sim 2 * 10\text{ms}$ / file

Very Bad Experience

- For 4,000,000 files, that's...
 - 22 hours