

CS 140: Operating Systems

Lecture 20: Caching and Performance

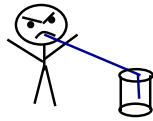
Mendel Rosenblum

Readings

- ◆ Readings (all on course web page):
 - A Fast File System for UNIX (paper)
 - Unix implementation paper

Sucking + blowing data through slow straws

- ◆ Latency tricks
 - Caching (data centric).
 - Code migration (code centric).
 - Prefetching (dual: asynchronous writes).
 - Clever data layout.
- ◆ Throughput tricks:
 - Increase bandwidth? Duplicate device N times.
 - Hide latency? Run another thread while waiting.
 - “Money can buy bandwidth. Latency requires bribing God.”
- ◆ These tricks are eternal themes
 - Use whenever need fast access to data living in slow place
 - Common straws: I/O bus, memory bus, network, space (bi-directional), time (uni-directional).



Caching: the buffer cache

- ◆ Our cliché: past predicts future? Use a cache.
-
- ◆ Buffer cache roughly similar to page cache
 - Good: memory growth on similar curve as disk capacity
 - Bad: forces 2 copies for normal interface; “cache wiping”
 - ◆ Our timeless cache questions:
 - How big? How to evict? What happens to writes? What to prefetch?

Decision #1: Size

- ◆ How big to make cache?
 - Main issue: partitioning memory buffer cache and VM page cache (its main competitor)
-
- Early systems: fixed-size cache.
 - ◆ Problem: doesn't adapt to workload.
 - Obvious sol'n: variable sized cache.
-
- Problem: enormous files not uncommon.
 - Solution: fixed size caches?
 - Same problem sharing page cache across “n” users.
 - Solve in same way (with the same slide!)

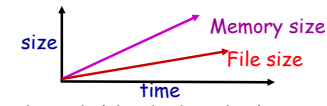
Split versus shared caches

- ◆ Private caches
 - Block cache and page cache have a separate pool of pages.
 - Miss in one can only replace one of its own pages.
 - Isolates cache, preventing interference.
-
- BUT: prevents one cache from using other's (comparatively) idle resources.
 - Efficient memory usage needs way to (slowly) change the allocations to each pool. Usually keep a fixed reserve.
 - Qs: What is “slowly”? How big a pool? When to migrate?

Decision #2: Eviction policy

- ◆ “Dance with the one that brought you”
 - The reason we used a cache was because past ~ future. So use LRU (as usual). New twist: can have perfect LRU!
 - New twist #2: Unfortunately, now it's less of a good idea since many files much bigger than VM objects.
 - ◆ What to do when file larger than cache?
 - LRU = exact worse thing. !LRU = best thing! (MRU)
- File has 4 blocks: 1 2 3 4
- | | | | |
|-------------------|--------------|------|--------------------------------|
| | LRU | MRU | |
| Reference strings | 1234 | 1234 | # of faults for 3 block cache? |
| | 123412341234 | | |

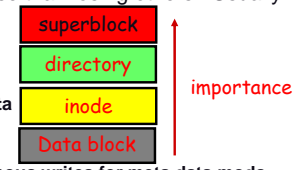
Decision #3: Write back policies

- ◆ Importance of writes?
 
- ◆ Write through (simple, but slow):
 - Whenever modify cached block, write block to disk.
 - Con: makes writes slow.
 - Pro: keeps FS in consistent state. PC OSes do this.
- ◆ Write back (faster, but complex (and dangerous!)):
 - Whenever modify block, mark as dirty. Flushed later.
 - Pro: fast writes, absorbs writes, enables batching
 - Con: More you defer write back, the worse a crash is.

Write back complications

- ◆ Fundamental tension:
 - On crash, all modified data in cache is lost.
 - Longer you postpone write back, the worst the damage is, but the faster you are.
- ◆ Four times to flush:
 - When block evicted - this is ~ what VM does.
 - When a file is closed - distributed file systems do this.
 - On an explicit flush - “man sync”.
 - When a time interval elapses - 30 seconds in Unix.
- ◆ Write back doesn't respect ordering
 - Crash = file in weird jumbled state.
- ◆ Finally: OS may not have any choice
 - Disk could be doing write buffering!

Flushing nuance: All blocks are not equal

- ◆ Losing some blocks worse than losing others. Usually:
 
- ◆ File system effects:
 - Flush modified meta data back quickly
 - Note: was an implicit side-effect of synchronous writes for meta data mods
- ◆ Application effects:
 - Have a really important file? Issue a manual flush (sync) to make sure its saved to disk.
 - Databases are forced to do this. Frequently, they will just by-pass file system.

Decision #4: what to prefetch?

- ◆ Optimal: the blocks we need are fetched in just enough time for us to use them.
 - Note: if we had enough disk bandwidth and ability to predict future wouldn't need (much of) a cache: just fetch block that will be used, and then throw way.
- ◆ Example:
 - App issues reads disk block every 1ms.
 - Disk can accept new request every 1ms.
 - Each request takes 10ms to satisfy.
 - How big a cache do we need?
- ◆ The usual problem: we don't know the future.
 - (Cache = trade space for stupidity)

Location ~ future (“Spatial locality”)

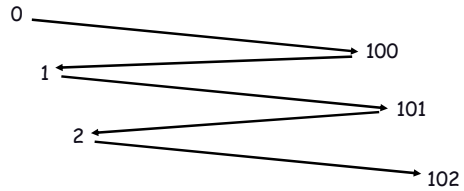
- ◆ “Spatial locality”
 - Access one object, will probably access ones close to it
 - E.g., read logical block in file, probably read next too.
 - So, when get one block, get the next n that follow.
 - More precise: when access one object, will probably access ones *related* to it. So, cluster them, and use above scheme.
- ◆ How big is “n”?
 - Tradeoff: larger n means more payoff if we are right, but more wastage if not. (Usually n < 65k bytes)
 - Variants: get big chunks & preferentially discard.

Decision #5: scheduling disk arm

- ◆ One resource, multiple requests = scheduling problem
One disk arm, multiple read and write requests.
- ◆ Goals?
 - Minimize seeks.
 - No starvation.
- ◆ Disk scheduling algorithms have close analogues in process scheduling
And, similarly, become more important as more requests. (However, unlike job queue, disk queue usually short...)
- ◆ Next: three scheduling variations
Optimize seek times.
Newer disks require taking account of rotational delay.

First-come-first-served

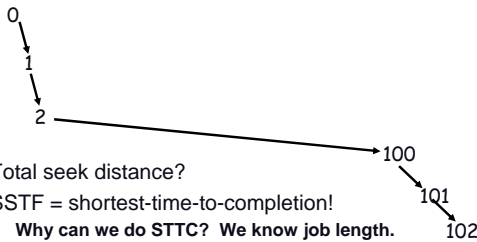
- ◆ Schedule disk requests in order received.
Fair but may have huge seeks for no good reason.
- ◆ Example: read from cylinders 0, 100, 1, 101, 2, 102



- ◆ total seek distance?

Shortest seek time first

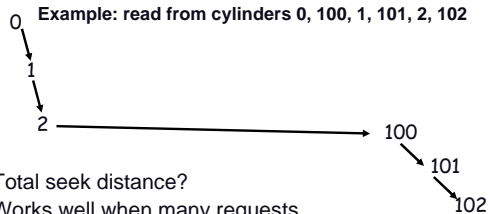
- ◆ Handle nearest request first
optimize disk arm movement, but can be (really) unfair
- ◆ Example: read from cylinders 0, 100, 1, 101, 2, 102



- ◆ Total seek distance?
- ◆ SSTF = shortest-time-to-completion!
Why can we do STTC? We know job length.

Scan (elevator algorithm)

- ◆ Move arm back & forth, handling requests underneath
more fair to requests scattered across disk, similar performance to SSTF



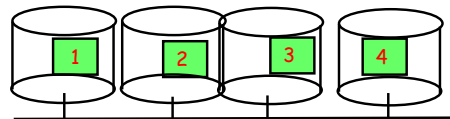
- ◆ Total seek distance?
- ◆ Works well when many requests
If not many, about 1/2 time won't get shortest seek

RAID: exploiting multiple disks

- ◆ "Redundant array of inexpensive disks"
(Acronym from same people that named RISC)
- ◆ Empirical observation:
To get given capacity much cheaper to buy n small disks than 1 large one.
Once you buy multiple disks, can do some neat tricks.
- ◆ Trick 1: n-fold bandwidth increase (ideal)
Stripe data across multiple disks.
Read/write from all simultaneously.
Common theme: money buys bandwidth by buying multiple straws - networks, disks, highway lanes,...
- ◆ Trick 2: use to increase reliability
Keep copies of data on different disks. Switch on failure.

Theme 1 in RAID: Major bandwidth

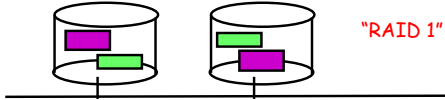
- ◆ Striping: smear data across all disks.
Ideal: N reads/writes going in parallel "RAID 0"



- ◆ Basic idea: straw too thin? Buy more of them and suck in parallel.
- ◆ Problem: reliability.
N disks, probability that one blows up increases as well.
Independent failures? 100 disks = 100 more likely that one is down. MTTF of 200K hrs/100 = 2Khrs ~ 3months

Theme 2 in RAID: Reliability

- ◆ Mirroring: improves reliability (and read performance)

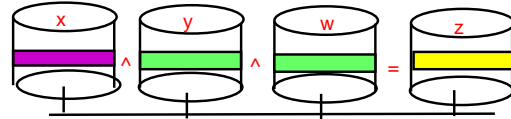


Every read/write operation issued to both disks.

- ◆ Basic idea (seen before): Duplicate state to increase reliability.
- ◆ Like backups except: Copy is completely up to data. Switchover is fast. Fault-tolerant systems use this basic trick a lot.

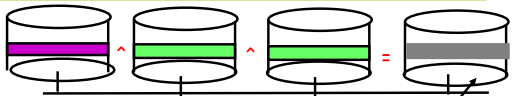
Cute xor tricks

- ◆ Recall: $x \oplus x = 0$
if $z = x \oplus y$ then: $x = z \oplus y$ and $y = z \oplus x$
proof: $x = x \oplus y \oplus y$, but $x \oplus y = z$ so $(x \oplus y) \oplus y = z \oplus y$
- ◆ So? Instead of mirroring, use one disk to hold z!
 $z = \text{xor of all other disks}$



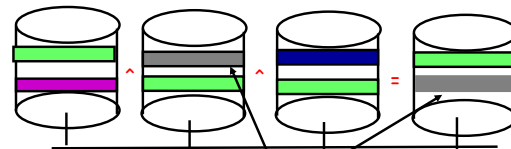
recover from any single failure by xoring the remaining disks with this "parity" disk.

Block-interleaved parity tradeoffs



- + less space than mirroring
- + much better bandwidth in ideal case
- ◆ Prob 1: "the small write problem"
Must recalculate parity on each write.
If write entire stripe, no problem. If not writing 1 block requires 2 reads + 2 writes!
- ◆ Prob 2: parity disk = bottleneck
If one disk used for parity, it will always be busy
sol'n: let the parity block in each group float around.

Block-interleaved Distributed parity



- + eliminates parity disk bottleneck
- + all disks can participate in read requests
best small read, large read, & large write performance (still whipped by mirroring on writes)