CSE 451

Journaling File Systems

Module 11

Block Cache

- Cache (often called *buffer cache*) is just part of system memory
- It's system-wide, shared by all processes
- Even a relatively small cache can be very effective
- File systems can "read-ahead" into the cache, increasing effectiveness even further
- Note that it is a **block cache**, not a file data cache
  - it operates below the level of the file systems

- Note: there can also be other caches, e.g., a directory cache
  - performance
  - uniformity (as part of VFS)

Block Cache and Write-Back

- Cache is generally write back
- Cache is meant to absorb many writes to a single/few blocks and turn them into a single IO operation
  - same with reads, but we interested in “delayed writes” here...
- File system “sync”s the cache based on time and activity
  - It also provides a sync operation for applications to call, if they want
- Updates since last sync are lost when system crashes

Writes, Caches, and Crashes

- Updates since last sync are lost when system crashes
- Not all writes are of application data
  - The file system needs to update metadata (in inodes, etc.)
- File creation example:
  - 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)
- Those four items are in four different disk blocks

Writes, Caches, and Crashes

- The file system itself may have consistency problems when a crash occurs between syncs
  - only some updated blocks make it to disk, but not others

- Example issues:
  - i-nodes and file blocks can get out of sync
    - files contain nonsense
  - block free map and and what the i-nodes claim can get out of sync
    - blocks are lost (not free but not on any i-node index)
    - blocks are both free and in some i-node's index
  - directory entry names an i-node whose updated contents were lost in crash
  - two different i-nodes index the same block(s)
  - etc.

- The problem is potentially much worse than losing the last few minutes of updates to files that were being worked on
  - If the file system metadata is corrupted, you can lose everything

Anticipating crashes

- Life has taught us we should assume things will go wrong...
  - *In this module we'll assume a block write either happens or it doesn't*
    - *A block is never written incorrectly*
  - *We'll also assume "fail-stop" behavior*

- Can I achieve robust updates by picking an order for the writes?
  - That is, is there an update order for which an arbitrary crash leaves the file system in a "not too bad" state?

- File creation example:
- 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)

- What order is right?

Can I Recover After A Crash? fsck

- File system may be in an inconsistent state
  - i-node map may indicate that an i-node is in use but no directory entry refers to it, or
  - directory entry may refer to an i-node that appears to be free, or
  - i-node may refer to data blocks that appear to be free (in the block map), or
  - data blocks may appear to be in use but aren't referenced by any i-node, or
  - a data block may be referenced by two or more i-nodes, or
  - ...

- **fsck**: Imagine writing a utility that scans the file system for consistency
  - all blocks not referenced in any way should be marked free
  - all inodes not referenced by any directory should be marked free
  - each data block in use should be used by exactly one i-node
  - i-node reference counts should be accurate
  - etc.

- Have to do these checks in "file order" not disk order
  - slow! really really slow!

- Have to apply heuristic recovery methods that may or may not work
  - All I know for sure is current values on the disk
  - I have to guess what values would have been had either the full update succeeded or none of it had been written

Journaling File Systems

- Goal: Make sure on-disk data is always in a consistent state
  - Note: A "consistent state" isn't the same as "no data has been lost"

- How?
  - update metadata [and, optionally, file data] ***transactionally***
    - *"all or nothing"*
    - *atomically*

- if a crash occurs, you may lose some work, but the file system structures on disk will be in a consistent state

- Achieve this by writing a "journal" of updates you intend to make before attempting the updates themselves
  - after a crash, quickly get it to a consistent state by using the transaction log/journal
  - cost is proportional to size of log, not the size of the disk

Where is the Data?

- In the file systems we have seen already, “the most recently written data” is in two places:
  - On disk
  - In in-memory caches

- In a journaling file system, the data may be in three places:
  - The in-memory cache
  - The “home copy” on disk
  - A journal entry on disk
  - (Note: The device may have its own cache, which complicates things...)

- The journal contains updates to be made to the home copy blocks
  - Note that those updates are in the in-memory cache so there’s no confusion if app accesses those blocks

Redo log

- Log: a chronologically ordered, 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
      - *log block "diffs" instead of full blocks*
    - this operation is idempotent
  - <commit t>
    - 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...

- Read and process 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.
    - Why does that matter? (It does!)
- Ignore uncommitted transactions
  - It's as though the crash occurred a tiny bit earlier…
  - Sure, you lose some work (updates), but the file system isn't corrupted

What about performance?

- Seems like you 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 in-memory 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

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
- As well as making recovery very efficient

Summary Questions

- What's the point of a journaling file system?
- Can data be lost if the system crashes?
  - Can data be lost if the disk device fails?
- Can file system meta-data be lost if the system crashes?
- What's the performance impact of journaling (and why)?
- Is journaling only for spinning disks?
  Only for SSDs?