CSE 451

Journaling File Systems
Module 10

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
- Even a relatively small cache can be very effective
- Many file systems "read-ahead" into the cache, increasing effectiveness even further

Cached Writes and Crashes

- Some applications assume data is on disk after a write
- The file system itself may have consistency problems if a crash occurs between syncs – i-nodes and file blocks can get out of sync
- 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)

Anticipating crashes

- Can I achieve robust updates by picking an order for the writes?
  - 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

- 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 this in “file order” not disk order
  - slow!

Journaling file systems

- 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
  - <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*
  - <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

- 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
- As well as making recovery very efficient