Lecture Notes - Filesystems: Caching and Consistency

Debadatta Mishra
Indian Institute of Technology Kanpur

Caching

- One of the commonly employed techniques to amortize the cost of disk access (which is a magnitude slower compared to memory access) is to cache the frequently accessed file data and the related metadata in memory. In absence of disk content caching mechanisms in memory, every access to disk (both read and write) must access the disk directly to serve the request resulting in sluggish application performance. In modern day file systems, O_DIRECT flag can be used in open() system call to perform I/O operations bypassing the cache layers.
- There are several advantages of DIRECT I/O in spite of the obvious performance overheads. They are,
    - Data can be directly copied from the user buffer to the disk to perform write( ) operations and copied to the user buffer from the disk to carry out read( ) operations. This avoids an additional copy operation required in cached I/O scenarios. However, the direct I/O operations require the user buffers to be block boundary aligned.
    - Without intermediate caching layers, the data consistency problem can be minimized by performing synchronous write operations on the disk by using O_SYNC flag along with O_DIRECT flag.
    - Many applications maintain their own memory caches (by allocating memory using OS memory allocation APIs) in order to utilize the caches in an efficient manner. The rationality of this design is well placed considering the applications are the best judge of their own access behavior. Some database systems employ application level caching and bypass the OS disk caches completely.
- There are several design considerations with respect to memory caching of disk content. First, the caching mechanisms require searching disk content in the cache. One solution can be to use the physical disk block number as the search key. This approach works in a file system transparent manner and can cache both file data and metadata. However, the file system level translation (file offset to the block number) must be carried out before searching the block in the cache. Another approach could be to maintain a per-file cache structure (a.k.a. Page cache) where the file combined with the file block offset is used as the key to search the disk cache. One of the advantage of this approach is that it does not require file system level translation before the cache access. While it may seem that it is not possible to cache metadata using the page cache, using the disk logical partition in place of the file inode can address this issue effectively.
- The second challenge with respect to cache design is to employ an efficient cache eviction technique to primarily maximize application level I/O throughput by improving the cache hit ratios. Many techniques explained in the context of swapping (also in CPU caches) are applicable in this scenario as the fundamental problem is same in

all the cases. Techniques like LRU, CLOCK, ARC etc. are most commonly used in most of the operating systems.

- The write policy employed in the caches has implications with respect to the performance and consistency aspects of the disk content. In writeback mode, any write operation is performed on the memory page containing the disk block followed by a delayed write to disk. The delayed write can be at the time of eviction or through periodic dirty block synchronization. Write back policy is most commonly used in operating systems (e.g., Linux) because it results in improved write performance. However, write back caches exacerbate the consistency issues. Another alternate policy is write through where the write to blocks are performed on both the memory page and the disk block at the same time. Write through policies result in poor performance and cannot effectively use the advantages of disk scheduling (e.g., reordering write operations to minimize disk seek latency).

Consistency

- File system consistency can be viewed from two different perspectives---(i) consistency from the user (i.e., system call) perspective, and, (ii) consistency from the file system perspective. Consistency with respect to system call API implies that any successful completion of a file related system call makes the operations promised by the system call persistent. For example, a successful completion of write( ) system call should ensure reading the newly written content irrespective of hardware/software failures occurring after the completion of write( ) system call. Similarly, successful completion of open( ) system call with O_CREAT flag implies the newly created file is accessible at any future time. Achieving user level consistency is particularly difficult in cached I/O scenarios with delayed write (write back) implementation. File system consistency refers to the sanity of file system data structures like inodes, block bitmaps etc. Depending on the file system semantics, there can be many invariants in the file system. For example, if a block pointer is pointed to by an inode, the used bit in block bitmap must be set to one for the corresponding block. Similarly, if an inode is contained as a valid directory entry, the inode must be a valid inode. Achieving file system level consistency is non-trivial and require special techniques.
- File system consistency issues arise when user operations translate to one or many disk write operations. If delayed/asynchronous write is implemented by the file system, even a write involving a single disk block update can result in consistency issues. With synchronous write, any user operation resulting in multiple disk writes can result in inconsistent file systems if a software/power failure occurs in between writes. Note that, most magnetic storage devices provide guarantees at the sector level i.e., any multi-sector operation does not provide any atomicity guarantees.
- Consider the file append( ) operation as an example where the data provided by the user (in a memory buffer) is appended to the end of file. Assume that, there is not enough space in the last block to expand the file and the file system allocates a new free block. Three different disk block updates are required to perform the above operation----(i) disk block containing the inode is updated to reflect the new size and map the new block through the inode pointer table, (ii) disk block containing the block bitmap is updated to mark the block as used and, (iii) disk block containing the user data. Note that, the inode block and the block bitmap are required to be brought to

memory before performing the write operation. For the data block, new data is written to memory before written to the disk, especially in delayed write scenarios. If the system crashes before all of the write operations are successfully completed, then there can be consistency issues as explained in the following scenarios.

- **Case 1 - Only the data block is written:** The file system consistency is not impacted even though the user does not find the appended block in the next read as the inode size and the data pointers are not updated. Therefore, the user level consistency can not be maintained in this case.
- **Case 2- Only the inode block is written:** The file system becomes inconsistent as the inode points to a location that contains garbage (data block is not updated). Even worse, the block used by the inode can be allocated for some other purpose (as the block bitmap is not updated) resulting in incorrect file system behavior.
- **Case 3 - Only the block bitmap is written:** The block bitmap update results in space leakage as the block neither contains valid data nor it is pointed to by any inode block pointers. This makes the file inconsistent as the total used blocks calculated using the block bitmap structure and sum of the used blocks calculated using the inode block pointers do not match,
- **Case 4 - The data block and the block bitmap are written:** Space leakage problem discussed in **Case 3** is also a problem in this case.
- **Case 5 - The data block and the inode block are written:** If the inode pointers (and size) are updated along with the data block, the user may temporarily observe correct behavior. However, as the block bitmap is not correctly updated, the same block can be allocated to some other file resulting in an inconsistent file system, just like in **Case 2**.
- **Case 6 - The inode block and the block bitmap are written:** Like in the **Case 1**, the file system remains consistent, but the user semantic of consistency is affected.

- File system consistency check (fsck) performs sanity checks on the file system data structures in cases when the file system is not cleanly unmounted because of software or hardware failures. The information related to the last FS unmount can be maintained in the on-disk superblock which is consulted during the next mount to invoke *fsck*, if required.
- The *fsck* utility performs checks to verify sanity of superblock, walks through the file system structures to establish correctness of invariants like the ones mentioned above. For example, fsck checks the block bitmap and the used block information from the inodes for file system consistency. While the file content cannot be verified, the directory content can be verified as it is structured by the file system. The major disadvantage of fsck is that it requires file system knowledge and require scanning the full file system.
- File system journaling is one of the commonly used techniques for file system consistency. Journaling is a concept similar to redo log concept used in database systems to implement transaction semantics in the databases. The idea of journaling is to create a note (write to journal) before the actual multi-step operation and use the note to redo the operations during recovery after the crash. The journal can be maintained in the same partition or on a different partition/device.

- For the append( ) operation explained before, the journal entry can be [START] [INODE BLOCK] [BLOCK BITMAP] [DATA BLOCK] [END]. After the journal write, the actual write operations are performed on the disk, possibly in a delayed manner. If all block writes are successful, the journal entry is marked as complete (checkpoint). If there is a failure in writing any of the blocks to the disk, then the journal entry is read during the next mount and all operations are performed to bring the file system to a consistent state. There can be failures during the journal write which is not a problem in a normal scenario (see below) as no operations are performed on the actual file system yet. Therefore, during the recovery, the FS can ignore the journal entry. One subtle issue may arise when the journal write is reordered by the disk scheduler where the [END] marker is written before (temporally) the data block. In such a scenario, the file system recovers the file to an incorrect state. To address this scenario, the [END] marker sector is written after all the other operations are journalled (written to the journal file/disk) provided that the [END] marker is written atomically.
- Implementation of journaling in file systems comes at the cost of huge performance overheads for normal write operations. Many file systems implement journaling only on metadata to strike a balance between performance and reliability aspects. One approach to implement metadata journaling can be to write the data before making the journal entry containing the metadata update information. This approach addresses the problem of file system consistency issues, but does not guarantee user level consistency semantics. For example, in the append( ) scenario explained above, the new data block is written before creating the journal entry containing updated inode and the updated block bitmap. In case of a failure during the actual write of the inode block or block bitmap, the file system can be recovered completely as the data is already uptodate. However, if the journal write fails, the file system is consistent but the user level consistency is affected.