User exposed methods

Regardless of implementation, in kernel, through FUSE or Dokan, a GUI application, or a base class in some programming language, the set of methods for manipulating files and directories is the same. Only one of the interfaces above needs to be truly implemented, as other can be mere wrappers.

Mounting a pool: User provides a list of paths to backends. Diskdict and pooldict is read from each listed backend. Newest valid value from all pooldict rings is taken as current. Each diskdict is checked against the random ids contained in the pooldict. This both sorts out which id goes to which path, and checks if all backends are present. If any backened mentioned in the pooldict is missing, mount fails, unless the user explicitly selected an override. Also some fields are checked for consistency, like if any segment was allocated more than once or if byte ranges are not overlapping. This is because this structure cannot be authenticated and malicious changes could be made when offline. Checksumming protects against accidental inconsistency. Pooldict is updated on each backend to current value, in case some rings were stale due to earlier interruption, unless mounted as read only. At this point mount is successful. File descriptors to backends are kept open, and current value of pooldict is kept in memory, until unmount.

Mounting a volume: Fsdict is loaded from all rings, and newest valid value is taken as current value. If the volume has access to many disks, then fsdict is duplicated in several rings, which are discoverable through pooldict. Each fsdict ring is updated to current value, in case some rings were stale due to earlier interruption, unless mounted as read only. If extent map is stored as entire dictionary, its chain is loaded and prepared for appending, if it is stored as B-tree, the root node is loaded. Last journal chain is loaded and prepared for appending. Mount is successful at this point. Current value of fsdict is kept in memory, until unmount.

Creating a file: Absolute path is computed. Root directory inode is loaded. Using its entry map, the inode for second path component is loaded. All inodes for intermediate path components are loaded. If any intermediate component does not exist in its parent directory or its inode has the wrong type, method fails. If the parent directory, which is the last intermediate inode, already contains an entry with the name, method fails. Revision number is recorded and incremented. New file inode is created, the revision number is assigned as the id. Journal entry is added. Inode is retained in memory and considered dirty.

Opening a file: Absolute path is computed. All inodes for path components are loaded. If any component does not exist in its parent directory or its inode has the wrong type, method fails. If atime is to be updated, the inode field is changed and inode is considered dirty. A new handle is assigned, perhaps a new revision number, and the handle is associated with the inode. Journal entry is added. Inode is retained in memory until closed and persisted on disk.

If path is like “/dirname/filename?rev=21” then file is opened as read only, chunk map is filtered so only chunks up to specified revision are visible, stat fields are computed from the chunk map rather than taken from inode fields directly, and inode becomes frozen at specified revision or completely.

Writing to a file: Open handle is used to find the inode in memory. Revision number is recorded and incremented. New chunk is added to the chunk map, with id set to revision number, buffer size set as the buffer provided, and a pointer to the buffer. Data buffer is stored in memory for further persistance and read operations, and is considered dirty. Overwrites do not discard previous buffers. Other inode fields such as newest file size and mtime are updated.

Closing a file: Open handle is used to find the inode in memory. If file was opened for particular revision, then inode is thawed for that revision. Handle is invalidated. Dirty inode and data are not persisted immediately and remain in memory.

Removing a file: Absolute path is computed. All inodes for path components are loaded. If any component does not exist in its parent directory or its inode has the wrong type, method fails. Revision number is recorded and incremented. Parent directory inode entry map is added a new entry assigning the filename to null id. File inode is not changed. Extent map is not changed. Journal entry is added. Parent directory inode is marked dirty and remains in memory.

Checkpointing: This is an internal operation called whenever few seconds have passed or enough dirty inodes or data accumulated in memory. For each dirty file inode, all dirty data chunks are processed by compressor. Fsdict active options dictate which compressor is used, if any. If chunks are too big, they can be either split before being compressed or let the compressor split the compressed stream. Second alternative is preferred but requires a specialized compressor implementation. Compressed chunks are concatenated into a data extent or few extents depending on available free space, each chunk followed by its checksum. Inode chunk map is updated with new extent id (new revision number for each data extent) and offsets/lengths within extent. Inode is serialized and also allocated disk space by fsdict, preferably right next to the last allocation. Extent map is updated for both extents. Extent info of previous inode blob is returned back to fsdict as unwiped. Dirty journal entries are allocated chunks within last journal chain. If latest journal chain is almost full, entries can be split apart. Both data extents, inode extents, extent map (chain or B-tree) chunks, and journal chunks are sent to disk. Disk is synced, and then fsdict is serialized and stored to its rings. There is no second sync.

Background process: This is an internal operation called every few seconds during peek hours and called continuously during night hours. End of the journal is scanned. If the latest entry is older than active options dictate, this entry is removed and the affected inode gets loaded and inode chunk map has entries dropped accordingly. If that removes one or all chunks from a data extent, the inode fields are updated to remember that. If data extent had all chunks dropped then the extent can be deallocated immediately. Inode id is put into the awaiting compaction dict, with the number of bytes expected to be freed as value. After the journal was finished, a second phase is commenced. Inodes from the awaiting compaction are loaded, and their data extents are moved and compacted, if they contain any dropped chunks. Extents can also be split and joined as part of defragmentation. Chunks can be recompressed if needed. Moved extents can be split or merged if needed. Extents are moved to a new segment. After all inodes were processed, third phase is commenced. Segments having most empty space or most small spaces are selected, loaded from disk, compacted and stored into new continuous segments. Extents can be split or merged if needed.

Conduits: Since FUSE does properly implement ioctl operations, not allowing data structures to be passed as parameters, alternative mechanism is going to be used for special operations. Each open() call having a path “?ioctl?” is going to return a descriptor connected to nothing. Writing to it represents a request with parameters passed as buffer, and blocks until that request completes. Reading from it returns the result of last operation. Several conduits can used concurrently, and each returns results of their requests.

[awaiting: creating snapshots, browsing revision history, reverting to a revision, ~~transactions~~, anonymous files, truncation, cloning files, ~~scrubbing process~~, ~~conduits for ioctl~~]