James Hong

Kuang Han

Time spent: 2 hours

The DeFiler file system will be implemented with three layers of abstraction: the file system layer (DFS), the caching and memory buffering layer (DBufferCache and DBuffer), and the block device layer (VirtualDisk).

The file system layer will manage the data in a high-level abstraction. This layer will be implemented as a single class called DFS, and will contain methods that allow the creation/deletion/listing of files, as well as read and write operations.

The caching and memory buffering layer will provide an interface between the raw block device and the file system abstraction. The buffers should reside completely in main memory, allowing for performance improvements since the number of necessary I/O operations will be reduced. In order to keep a constant size buffer with LRU replacement policy, two data structures were considered: an array with a timestamp field for each entry and a linked list sorted by recent usage (most recent on top of the list). The array structure could provide O(1) access to each buffer and O(n) time for replacement of a single element (due to necessity to scan entire array for oldest timestamp). The linked list could provide O(n) access to buffers and O(1) time for replacement. It was decided that the linked list would be the best data structure to use in this case because the potential worst case scenario for replacing all elements (such as when using the buffer for data streaming) is O(nˆ2), which could cause bottlenecks in performance.

An LRU replacement policy was decided as the best option in this case because it seems to have high performance for the majority of use cases. It only performs badly when performing large sequential accesses but then, those types of accesses are hard to cache with any replacement policy. Only prefetching would improve it. Software allows us to use a true LRU (instead of pseudo LRU, which is actually NMRU, as implemented in hardware) with high efficiency.

The raw block device layer is implemented in the class VirtualDisk, and since it interfaces directly with a file, it should not require implementation of a data structure other than the file handler.

The thread model in this file system is that every concurrent I/O operation uses a distinct thread. A thread that originates somewhere in the application and wants to perform some I/O on a DFile will call a method at the top layer of the file system (the DFS), which will in turn call the middle layer of buffers and cache. Depending on the cache status and eviction policy, the I/O operation might go to disk and call the lower layer (VirtualDisk). The methods that are actually called depend on the operation performed, but the flow is intuitive: a read() on the top layer will invariably call read() on the middle layer (if the buffer is present and valid), otherwise, the startRequest() asynchronous method in the VirtualDisk gets called and the data contents get loaded to a DBuffer. Notet that the interface between the DBuffers and the VirtualDisk is asynchronous: the method returns shortly after being called and the completion of the task is signaled by callbacks to the middle layer. Some methods are special for bookkeeping and should be called in some circumstances. For instance, sync() in the DBufferCache should be called before exiting an application in order to flush the buffers to disk.

In terms of synchronizing accesses, basically both the DBufferCache and DBuffer need to have block mechanism inside. DBufferCache will have a lock on its DBuffer list (or set depending on the implementation) so that any get/release call on the buffer list will be atomic. This is to prevent the case that eviction and access calls happen on the same buffer. Another way of implement this is to create a lock for each buffer in the cache, so that the access to each individual cache slot will be atomic. Both of the two solutions should work, and we need to figure out which way to go during the implementation.

Each DBuffer instance also needs a lock. The lock will be on its internal byte array so that any read/write/IO call will be atomic. This will solve the issue that when one process is reading or writing data in a buffer, other processes will be blocked evicting the block (because evicting the block requires disk IO).

We plan to test our code using stress tests that are based in common I/O scenarios: reading and writing large files from/to disk sequentially, performing random accesses in a large file, performing strided accesses in a large file, performing random accesses in multiple small files and performing concurrent random reads and writes (including deletion). Also, bookkeeping test cases will be implemented, checking if the creation/deletion of a file is successful and behaves as expected, and warning the user if a VDF has gone corrupt. Note that random accesses are not a standard feature of the file system, however, they are necessary for most applications. Therefore, random accesses will be simulated by ignoring the bytes that are seeked (or in the case of a write, reading and then writing to the same location until the desired location is reached, when the contents to be written will be replaced to the application’s desire), with potentially disastrous performance. This list is not exhaustive and the actual test suite could include many more other situations. Using java’s timing methods, the performance of those operations as a function of the parameters in the file system could be recorded and analyzed.