Course Work 3

High-level design:

API:

LRUCache(int capacity):
Constructor to initialize the cache with a given capacity.

bool contains(int offset, void* buffer):

Returns true if block is present within the cache and copies data to the address pointed to by the buffer pointer
@param buffer pointer to address of buffer.
@param offset used as a key for the map in the cache.
@return Returns true if block is present within the cache

void addLRU(int offset, void* buffer):
Essentially creates a key, value pair where the key is the offset and value is the pointer to the node which contains the data at the buffer address.
We copy the data at the buffer to this node so that we can use it later in the cache with O(1) lookup. If on adding, the size exceeds the capacity, we use an LRU policy to get rid of an item from the tail of the dll.
@param buffer pointer to address of buffer.
@param offset used as a key for the map in the cache.
@return void

void makeMostRecent(Node* node):
Moves the node to the head of the list, essentially making it the most recently used node in the dll.
@param node – node of class node
@return void

Data Structures:

Doubly-linked list (DLL) to store cache entries as nodes. This allows for efficient insertion and removal operations at both ends (head and tail) of the list.
HashMap to store the cache entries for quick look-up by offset. This ensures constant-time access to cache entries. The key, value pairs are offset, pointer to a node, where the node contains the required cached data.
-Node
class Node is a class used to create nodes of a dll. offset contains the offset provided in the read_block fn, we use isValid to bypass the restriction of not having a map.remove(), where although we don't remove the key from the map, we remove its node from the dll and make it invalid if it exceeds the given capacity and data is a pointer to the address which will contain the data to be cached.

The cache uses a Least Recently Used (LRU) eviction policy to make room for new entries when the cache is full.

Performance complexities:

Because we use a map to access the node of the dll, we have O(1) lookup with the dll as opposed to a O(n) lookup.

Read path: O(1) for cache look-up to the node and because of this, we need O(1) for moving a cache hit entry to the head of the DLL as opposed to using a linear search. Overall complexity is O(1).
Eviction policy: O(1) for removing the LRU entry from the tail of the DLL and O(1) for adding a new entry to the head of the DLL. Overall complexity is O(1).

Correctness/Consistency:

The cache ensures correctness and consistency by:

Properly initializing the cache and its data structures in the constructor.
Using the contains() method to return to the buffer address the cached data if it is a cache hit or calling transfer otherwise.
Using the addLRU() method to update the cache with the correct data after a successful transfer.
Using the moveToHead() method to maintain the LRU order in the cache after each access.
Always returning the correct data from the cache or performing a transfer when a cache miss occurs.
Also, since the cw is a read-only buffer, consistency is guaranteed as the data will not be altered by any writes.

Limitations:

Performance: The current implementation may suffer from performance issues when dealing with large datasets. The current design does not support map.remove , so it may lead to indefinite growth of the map, but the dll never exceeds the given capacity.

Space: the use of a doubly-linked list and HashMap for cache management adds some overhead in terms of space consumption, and for larger datasets, the map can grow indefinitely as there is no remove function which will have a negative impact on the space complexity.

Advanced Section

Running Instructions: In order to run the advanced implementation, please replace the addLRU function with the addMRU function in the read_block function. Since both functions are defined in page-cache, this is all that is needed to run it.

API Details

void addMRU(int offset, void* buffer):
Essentially creates a key, value pair where the key is the offset and value is the pointer to the node which contains the data at the buffer address.
We copy the data at the buffer to this node so that we can use it later in the cache with O(1) lookup. If on adding, the size exceeds the capacity, we use an MRU policy to get rid of an item from the head of the dll.
@param buffer pointer to address of buffer.
@param offset used as a key for the map in the cache.
@return void

For the advanced section, I have implemented a cache which uses the most recently used (MRU) eviction policy instead of the LRU policy. In order to run it, all we have to do is change the add function in the read_block function to addMRU. It works by removing the head as opposed to the tail of dll when capacity is reached.

The MRU acts by removing the most recently used element, based on the idea that if you just saw something, you're less likely to see it again, which is useful under conditions where the older an item is, the more likely it is to be accessed, for example in a ticket booking software, you would want to get rid of the most recently booked ticket from the cache.

For the testing, I ran the README twice using both algorithms. Both Caches were initialized with a capacity of 64.
On the second try for the LRU, I encountered 0 Cache Misses, whereas for the MRU, I happened to encounter a few Cache Misses. So for this case, an LRU works better as we are accessing the recently used items quite frequently, and an MRU would work better for cases where the older an item is, the higher chance it has to be accessed. An MRU would work much better for certain looping behaviors [Chou], for example, if we had 65 blocks we had to read twice in the same order, in the first run, both Caches would have 65 misses, and the LRU would have blocks 2-65, whereas the MRU would have 1-63 and 65. So on the second run, the LRU would only have cache misses, whereas the MRU would have no cache misses until it reaches the 64th block, which is a much better performance than an LRU cache.

The choice between LRU and MRU eviction policies for a cache depends on the specific use case and the characteristics of the data being cached. There is no one size fits all approach when it comes to caching, and the efficiency of a cache implementation depends on the specific context in which it is used. For example, if the data being cached has a high likelihood of being accessed again soon after being accessed, an LRU cache may be more efficient. On the other hand, if the data has a higher likelihood of being accessed again after a longer period of time, an MRU cache may be more efficient. Therefore, the choice of eviction policy for a cache should be made based on the specific requirements and characteristics of the data being cached.

References

Hong-Tai Chou and David J. DeWitt. 1985. An evaluation of buffer management strategies for relational database systems. In Proceedings of the 11th international conference on Very Large Data Bases - Volume 11 (VLDB '85). VLDB Endowment, 127–141.