OS CW3 Report

High-level design of the cache

The API design for the basic part is best illustrated in the page-cache.h file. In there, I have created two things: a ***struct***, which is used to represent individual blocks within the cache, as well as a ***class*** to represent the cache itself.

The struct for the blocks keeps track of the following information: the ***block number/ ID*** (this metric is used to identify specific blocks within the cache), the ***contents of the block*** (the actual data), as well as ***two pointers***, one to the ***previous*** block, and one to the ***next*** block in the cache.

The reason for the two pointers is due to the data structure used to represent the cache, which is a ***doubly linked list*** [1]. The doubly linked list is used to keep track of which block in the cache needs to be evicted in accordance with the Least Recently Used (LRU) policy [2].

The actual skeleton for the cache class consists of three methods: ***initialise*** (which creates the initial cache, with the base values for the current size of the cache as 0, and the first and last elements set as NULL), ***retrieve*** (which is used to read a block if present in the cache), and ***insert*** (which implements the LRU replacement policy.

Changes to the ata-device.h and ata-device.cpp files

The main change inside the ***ata-device.h*** file was the addition of a private variable for the cache. This was done by adding an include statement with the appropriate header file, such that the cache class was recognised as a data type, and then simply adding a variable amongst the other private variables in the ATADevice class.

As for the ***ata-device.cpp*** file, the main changes included once again adding an include statement with the appropriate header file, such that the cache class was recognised as a data type, as well as calling the variable's ***initialise()*** method inside the ***init()*** method in the cpp file. Lastly, the most major change was made to the ***read_blocks()*** method: here, we first check if the block is contained within the cache, in which case we report a hit; secondly, if we do not find the block, but we can transfer the block from memory, we report a miss, but still return true (as the block could be read); lastly, if both previous conditions failed, then the block cannot be read, in which case we return false.

Performance complexity of read and eviction policy

The performance complexity of the read and eviction policy are highly dependent on the ***doubly linked list*** data structure used [3].

For the ***read*** operation (which is performed by the ***retrieve()*** method), the time complexity would be O(n), and the space complexity of the operation itself would be O(1). This is because, in order to read a certain block, a list traversal is necessary to first identify where the block is in the list (this operation requires O(n) time). Once the block is identified, the remaining operations each have a time complexity of O(1). Therefore, since the list traversal takes up the most significant amount of time, ***the complexity of the read operation is O(n)***.

As for the ***eviction policy*** (which is performed by the ***insert()*** method), the time complexity would be O(1), and the space complexity of the operation itself would once again be O(1) [4]. This is because, for a deletion to occur, only one block is removed from the linked list, with access to this block not requiring a list traversal. Therefore, since the complexity of deletion from a doubly linked list is O(1), ***the complexity of the eviction policy is O(1)***.

How to ensure correctness/ consistency in the cache

The cache ensures consistency and correctness by constantly updating the doubly linked list every time a retrieve or insert call is made. If a read call is made, the list is updated such that the block that was least recently accessed (which is the last element of the list) gets moved to the front of the list in case of eviction.

When performing an insertion, two cases must be considered: first, if the cache is not full; second, if the cache is full. In the first case, we simply insert the element at the front of the list since it represents the most recently used block. In the second case, however, we overwrite the last block in the list (since that will automatically be the least recently used item) and move this altered block to the front of the list.

To illustrate why this works, let us consider the following example. First, several caches are inserted into an empty cache until the cache becomes full. In this case, the list is ordered with the least recently used item at the end (the last element). Therefore, if we were to perform another insertion, the least recently used block would be replaced (as expected). If, however, we perform a retrieve operation for the first block we inserted into the cache, this block will then be moved to the front of the list, therefore pushing all other elements up (incrementing their index by 1). If we were now to perform an insertion, the block that was initially inserted second in the cache will be overwritten (which is once again as expected, since that is now the least recently used block). This cycle can be repeated multiple times, and as can be seen, the condition for the LRU eviction policy always holds: the least recently used block is always evicted.

Limitations of this approach

Once again, due to the ***doubly linked list*** data structure used, the main limitation is the space complexity of the cache, which is O(n). This means that, if we were to have a cache that is fairly large, this would require a significant amount of memory space for the current implementation to function as intended. Additionally, due to the two pointers used (one to the next block and one to the previous block), this implementation is also less efficient than using a simpler single linked list or circular linked list [5].

Comparison of the two policies (advanced part)

The two policies being compared are the Least Recently Used (LRU) and the First In First Out (FIFO) eviction policies. The LRU policy requires keeping track of when each of the blocks was last accessed, and evicting the one that has gone the longest amount of time without being accessed. Meanwhile, the FIFO policy simply evicts blocks in the order in which they were added, with no regards to how often they are accessed [6].

The main performance metric used to compare the two policies was the cache miss rate on the first attempt of reading different files. The main thing noticed was that the LRU policy performed better than the FIFO policy on all files, with some files offering very minimal differences (e.g., one miss), whereas other files offering more substantial differences (e.g., 5+ misses). For example, the available README file provided a minimal difference: the LRU cache only incurred 2 misses on the first attempt, whereas the FIFO cache incurred 3 missed on its first attempt. However, it is worth noting that, in all test cases, both caches were able to achieve 0 misses on the second attempt.

However, these results are to be expected: LRU aims to keep the blocks that are most recently used within the cache, whereas FIFO keeps the blocks that were most recently added. If, for example, we were to insert 512 blocks in the cache (the maximum size for the current implementation), both caches would be full. If we then performed a read on the first 3 caches, this would have no impact on the order in the FIFO cache, but it would alter the order of the LRU cache. If we then wanted to insert a new cache, and then read the first cache we initially inserted, the LRU cache would see a hit, as the block is still stored there, whereas the FIFO cache would see a miss and would have to fetch the block from memory [7].

References

[1] https://en.wikipedia.org/wiki/Doubly_linked_list

[2] https://www.educative.io/answers/what-is-the-least-recently-used-page-replacement-algorithm

[3] https://www.programiz.com/dsa/doubly-linked-list

[4] https://www.interviewcake.com/concept/java/lru-cache

[5] https://www.javatpoint.com/singly-linked-list-vs-doubly-linked-list

[6] https://en.wikipedia.org/wiki/Cache_replacement_policies

[7] https://link.springer.com/article/10.1007/PL00009255