CSE 303: Quiz #5

Due November 13, 2019 at 1:35 PM

Please use the remainder of this page to provide your answer. To submit your answer, create a pdf whose name is exactly your user Id and the "pdf" extension (e.g., abc123.pdf) and email it to spear@lehigh.edu before the deadline.

1) The LRU policy for cache replacement is known as being a "stack" algorithm. This means, among other things, that it is immune to "Belady's Anomaly". What is Belady's Anomaly? What does it mean for LRU to be immune to it?

Belady's anomaly is the idea that as you can hold more page frames, you might actually get more page faults depending on the data you're given. For example, if you can hold 3 page frames, you might get 8 page faults, whereas if you hold 5 page frames, you might have 10 page faults. LRU is immune to it because it is a stack algorithm. This means that the smaller set of stack frames will always be a subset of the bigger set of stack frames, even as things move in and out. You have 3 situations that might arise once both stack fames are full (we assume that the smaller (S) one is size n and the bigger (B) is size n + 1 and the first n of both are the same). Case 1: you call for something in both S and B. This element moves to the top of both S and B, and the first N remain the same. Case 2: Both get page faults. You evict from the bottom of S, it's equivalent moves into B's "extra spot," you add the same element to the top of S and B and everything shifts down one. S is still a subset of B. Case 3: S has a page fault, B doesn't. This means you're calling for B's "extra spot." This spot moves to the top and everything else shifts down one, while something new moves into S and everything shifts down one. Thus, the structure is maintained. If everything in S is also in B, we can't have a page fault in B and not in S.

2) In class, we discussed how it can be difficult to implement LRU efficiently in software. What are some problems that you see, and how do you think one could work around them? You might want to describe a specific scenario, since LRU is applied differently at different parts of the system. (Please focus on software, not hardware).

For LRU, we need to be able to 1)Always know our least recently used element for when we evict, determine if an element is in the set before we choose to add, remove an element from the set before we add, and add. From a big O perspective, the we can do everything in O(1) time: keep a doubly linked list of our objects based on time and a hashmap mapping from a key to a node. We can insert into the doubly linked list in O(1), remove in O(1), and update in O(1) (find our node from the hashmap and remove ourselves and then add to end of linked list). The issue here is if we're using a multithreaded algorithm, we risk deadlock: to remove from a doubly linked list, we need locks on the node we're deleting and the node before and after it. Thus if we consider them nodes 1, 2, and 3, if I grab a lock on Node 2, but then someone tries to delete node 2, but someone else is trying to delete node 1 and grabs that lock, both deletes are waiting on locks from another thread and we hit deadlock. This application would thus work best in a single threaded application where n is huge.

If n is small, we could use an array and every time we do something, we search the array to see if it's in there: if it is, we update the value, if it isn't we search the array for the smallest timestamp. This would be O(n) time. To update a timestamp, you would only need 1 lock and wouldn't risk deadlock. Evicting a node might be problematic because once we find the LRU element, another thread could've already evicted it. However, there are solutions to this such as holding a lock on the smallest element we've come across so far and when we find a smaller one, lock it, then unlock the previous smaller item's lock once we have the lock and confirm it's smaller. This might work better for multithreaded applications where n is small since we do have the O(n) time for everything.