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 phenomenon where increasing the number of page frames could also increase the number of page faults in a cache. Normally one would think that adding more frames allows for a wider coverage of the physical memory, but depending on the algorithm used to replace the cache on page faults, you could result in Belady's Anomaly taking affect. FIFO is an example of an algorithm where the phenomenon can occur because it is not a stack algorithm. A stack algorithm means that the page frames of a cache will always be a subset of the cache with additional frames. Because of this property, the cache in FIFO could end up in different states depending on size that could lead to the anomaly. However, in LRU any smaller cache will be a subset of the larger ones because they will both contain a stack of the most recently accessed pages. There is no scenario where the smaller cache will hit without the larger one also hitting, so Belady's Anomaly cannot occur and makes LRU immune.

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)

It is hard to implement LRU efficiently in software because you need to choose a data structure and method of conveying when something was last accessed while also keeping in mind the use of concurrency. One way of implementing LRU is through a queue represented a doubly linked list. You add pages to one end and remove from the other. What makes this usable for LRU is that there is some sense of order and when a miss occurs you can just quickly pop and add the needed page. Whenever a hit is made you can just move the element that was hit back to the end of the queue. As discussed in class however, this might not be the most efficient method. The main issues arise when you start thinking about concurrency and run time for each operation. The runtime of additions and removals are very fast, but in doing this the program is optimized towards removals, so working with hits requires a lot more effort because there would be a lot of data movement and iteration. Since there is a lot of data movement, it also affects concurrency because more locks would have to be acquired. Another strategy for implementation is making use of hashes. Using hashes allows for a quick lookup when there is a hit through the hash. The downside of this method, however, is that there is a larger space requirement to keep track of time stamps and the map itself. Also, reverses towards being very inefficient when it comes to removal from the hash. The algorithm would need to go through the map and compare timestamps to remove the oldest. This method also has the benefit of touching less data and no data movement when hits occur, which makes concurrency a little easier.