Thomas Chung

COEN 177

Page Replacement/Caching Simulation Lab

In this lab, a cache simulation was use to test out the outcomes of four different paging algorithm: Least Frequently Used (LFU), Least Recently Used (LRU), Second Chance (SC), and Random (R). All four of these programs used a double ended queue (deque) to keep track of the pages that were being passed in. They were implemented as such:

* **LRU**: The program checks to see if the number is already in the queue; if it is, then move it to the front. If it isn’t, it’s a page fault. If the deque is full, then pop out the back and put this new number in front. If not, just add this number to the front. This ensures that the most frequently used page is at the head of the deque and allows for easy access.
* **LFU**: Each page that was passed in invoked a Page class object to be created for them that held two variables: their number and their number of use. Every time a page is passed in, the program checks to see if it already resides in the deque. If it is, its number of use gets incremented and the Page is moved to the front of the deque. If not, we check if the deque is full. If it is not full, we push the new page onto the front of the deque. But if it is full, the program linearly runs through the deque to find the Page with the lowest amount of use and removes it. The new page is then push to the front of the deque.
* **Second Chance**: Each page that was passed in invoked a Page class object to be created for them that held two variables: their number and a Boolean called chanceBit. The chanceBit refers to the ‘second chance’ aspect of the algorithm. If a page that was passed in that was not found in the deque and the deque is full, then the program examines the head of the deque for a Page with its chanceBit set as false. If the head has a chanceBit as true, then we move the head to the tail and set its chanceBit as false. When the program finds a page without a chanceBit as true, it removes it and adds in the incoming new page to tail. If the incoming page was found in the deque, then the page’s chanceBit is upgraded to true.
* **Random**: Random is implemented similarly to Second Chance. When the deque is full, it looks for a page that no longer has a second chance with the same process SC uses and purges it from the deque. It then inserts the new Page somewhere randomly into the deck.

To test the accuracy of each paging algorithm, we first inputted 10 random page numbers and determined by hand how many page faults that would be. We then ran it into the program and saw that it was correct. To test for other memory sizes, we were given a ‘numbers.txt’ that had a list of over 10,000 numbers along with the correct solutions to each paging algorithm. We ran our programs for memory sizes of 10-500. The following graph is a plot of the hit rates for the different algorithms that were implemented:

What’s interesting to note is that LRU and Second Chance have very similar hit rate due to the nature of their algorithm to maintain order by moving the most recently used to the front of the deque. Random and Second Chance have very similar hit rates because they were implemented very similarly as well, with Random using Second Chance’s way of iterating through to find a Page without a chanceBit on.

To optimize these paging algorithms, I believe it would be best to use a more dynamic cache to hold the Page files along with a better search algorithm. In this lab, a deque was used because it had O(1) of insertion rate and O(n) search time. A singly-linked list would have O(n) insertion time but could use other searches like binary search, DFS, or BFS.