Felicia Kuan

COEN 177

README for Lab 4

To make the functions simpler, I placed the parseArguments and isInMemory functions in the file “init\_read.h”. parseArguments parses what the user inputted into the program and isInMemory returns the index of the page within the pageTable. If the page is not found, it returns -1.

I also made a “helper\_fxns.h” to put a find the index of the page with the smallest recency counter.

We defined hit rate to be the (number of requests - number of misses) / number of requests.

**Explanation for FIFO:**

The FIFO algorithm in general saves the page value in the table, and once the table runs out of space, the algorithm kicks out the page that first entered the table.

For my FIFO algorithm, if the number of page requests have not yet exceeded the size of the page table, the page request is saved at an index, called pageTableIndex, and the index is incremented after. If the index has exceeded the size of the page table, I saved the new page at the spot in the pageTable array where the index is mod-ed by the size of the table. I also incremented the index afterwards too.

Since it does not matter to the FIFO algorithm whether the value is in the table, that is all my algorithm did .

**Explanation for LRU:**

In general, the LRU algorithm stores pages in a table and when the table fills up and a new page wishes to be stored in the table, LRU replaces the page that is least used.

To implement this, I created a global counter called “recency counter.” At the beginning of the program, an additional array is created called LRU\_cache that’s the same size as the page table. This array will store the recency counter, which keeps time, and will tell us how “recently accessed” (quantified by the value of recency counter) the page that is stored at the same index is.

While a new page is to be added and the table has not yet filled up, the new page is stored in the table at pageTableIndex. At the **same** index of LRU\_cache, the array stores whatever the value recency counter is at the time. Then, recency counter and pageTableIndex is incremented.

Once the page table fills up, one of two things can happen: the page the user is trying to store (pageRequest) is 1) in memory or 2) not in memory.

1. If the page is in memory, we simply find the index of the page (isInMemory tells us this), go to the page’s corresponding index in LRU\_cache and update the value to the new recency counter. Afterwards, recency counter is incremented.
2. If the page is not in memory, we have to find the page within the array that has the smallest recency counter value, which is stored in LRU\_cache. Here, I made a function min that returns the index of the page that has the smallest value in LRU\_cache. Small value in LRU cache means the page is the one that was accessed furthest in the past compared to the other pages. The new page the user is adding in is saved at the spot (index) found by the function min.

**Explanation for Second Chance:**

In general, the second chance algorithm is like first in first out, but implements a flag (which I called ref) that if a page had a flag raised, the page gets a chance to be kept within the page table for one more round.

To implement this, I created a struct that contained a page\_value, which is where the pageRequest will be stored, and a reference bit. It basically worked like a pair, where page\_value stored the page, and ref was the binary bit that stored the flag.

While the page table isn’t full, the pageRequests are stored in the table where the pageTableIndex tells them to be. Also, the corresponding flag (reference bit) is set to 0 and pageTable Index is incremented, essentially placing the new page at the back of the line.

When a new page is added and the page already exists inside the table, the algorithm sets the reference bit of that page to 1, allowing it to live through another cycle. This is achieved by the isInMemory2 function, which behaves just like isInMemory by returning the index of the page in the array/table that matches the page request. The only difference is that isInMemory2 works for the struct.

Once the page table fills up, the algorithm goes through and looks for the first page whose ref bit’s value is 0 and replaces that value with the new, incoming page. If the algorithm runs into any pages whose reference bits contain the value of 1, it sets the reference bit to 0, spares it, and continues forward to look for a page whose reference bit value is 0 to be replaced.

To compile:

gcc helper\_fxns.h helper\_fxns.cpp init\_read.h init\_read.cpp FIFO.cpp -o FIFO.o

gcc helper\_fxns.h helper\_fxns.cpp init\_read.h init\_read.cpp LRU.cpp -o LRU.o

gcc helper\_fxns.h helper\_fxns.cpp init\_read.h init\_read.cpp sc.cpp -o sc.o

to run: cat

./TestResults/accesses.txt | ./FIFO.o 100 >> FIFO\_results.txt 2>&1

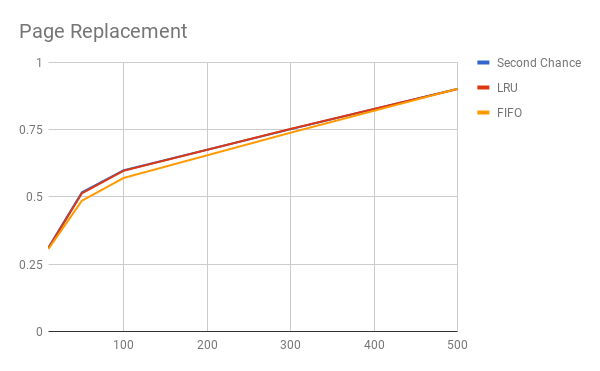
cat ./TestResults/accesses.txt | ./LRU.o 100 >> LRU\_results.txt 2>&1

to compare: diff FIFO\_results.txt ./accesses.txt

The following are my results, displayed on two graphs and a table, of my hit rates for the three algorithms:

The leftmost column represents the page tableSize.

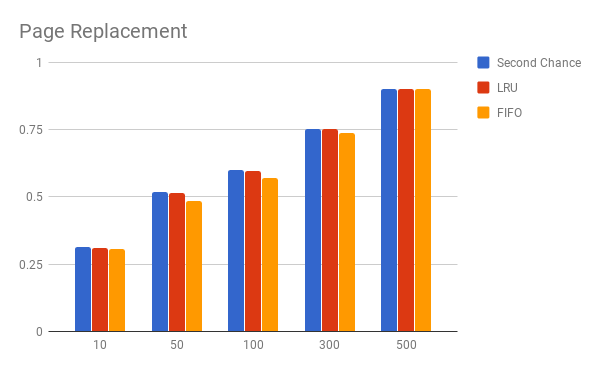
|  |  |  |  |
| --- | --- | --- | --- |
|  | Second Chance | LRU | FIFO |
| 10 | 0.312724 | 0.31015 | 0.306953 |
| 50 | 0.517429 | 0.513629 | 0.486111 |
| 100 | 0.599303 | 0.597564 | 0.570789 |
| 300 | 0.752037 | 0.753123 | 0.738997 |
| 500 | 0.90199 | 0.901282 | 0.902122 |



Blue: Second Chance

Red: LRU

Yellow : FIFO



Observations:

In general, especially when the page table sizes are smaller, FIFO has a lower hit rate than the other two algorithms. In contrast, second chance performs the best, and LRU performs just a little below Second chance.

This trend continues all the way until the page table size reached 500, and at this point, FIFO actually performed better than both second chance and LRU. It only performed slightly better, but it is enough to notice that FIFO caught up to the other two.

This shows that how well these algorithms perform depend on the tableSize, and even though all of them perform approximately just as well, there is a “better” algorithm that has the highest hit rates at each bracket of the table size.

And this makes sense. When the table sizes are smaller, FIFO performs worse than the others because it is arbitrarily kicking all the pages out without much thought. Statistically, when the table size is small, the new page you’re looking to insert into the table is less likely to be already in the table than when the table size is much larger, such as at 500. Thus, the larger the table size, the better FIFO will perform to increase the statistical probability of finding the page you’re looking for inside the table and increasing the hit rate.

According to our data, it seems that LRU and Second chance have about the same performance according to our metric of evaluating the performance of the algorithms by hit rate.