Justin Li

3/4/21

**Lab 8: Memory Management**

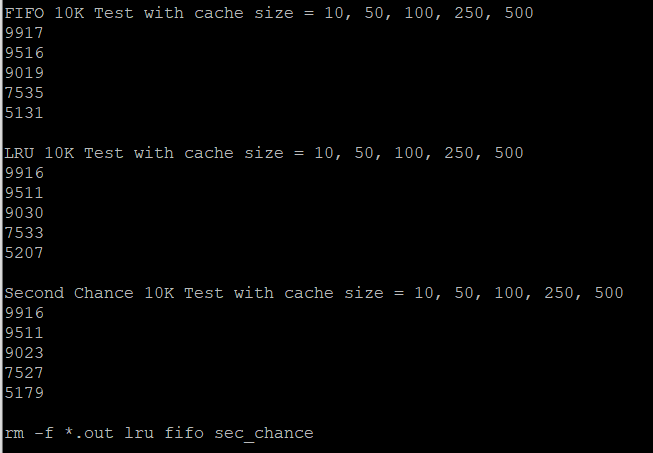
1. Write up a description of your implementations and **sample miss-rate (page fault rate)** results, and submit it alongside your code. This portion of the assignment is as critical, if not more so, than the actual implementation of your solution.

For all 3 algorithms, I followed a relatively similar structure of organization in my programs for the respective page replacement algorithms, having slight differences between them. First, I created the typedef struct{}, and included an integer variable to hold the page number. In both LRU and second-chance, however, I additionally added an integer variable to store the index as well, one to keep track of the LRU, and the other to keep track of the second-chance marking. In the main() function for all 3, I included the size of the cache passed by the user, the cache itself set to that size, and a cache to hold the page inputs from the test files, with size 100. I then defined the arbitrary integer variables as needed, and created an integer variable totalFaults, keeping track of the total amount of page faults for each algorithm. I then used a for loop to initialize each of the struct variables in each program. Then, I added the while(fgets()) functions, under which I implemented the respective page replacement algorithms. I first stored each number read from the file in my page\_num integer variable. Then, I created a foundInCache boolean variable, which I used in a following traversal for loop. In the for loop, I check to see if the page number is in the cache. If it is, then I update foundInCache and the indices of the pages in the cache and break out of the loop. However, if they are not, then after the for loop finishes, I check to see if foundInCache is false, and if so, the program then runs the following if-statement. In all 3, I begin by printing which page caused the page fault, replace an old page with the new page in the cache, and increment totalFaults. In FIFO, I use the oldest variable to mark the location of the oldest page, which I replace with the new page, and increment after each replacement. Once it reaches max size of the buffer, I then wrap the value back to 0. In LRU, I update each index by incrementing when not foundInCache, find the least recent element, mark the position, then replace. When the page is foundInCache, we do a for loop through the whole cache, and increment only the pages with a lower index than the currently found page. Finally, for second-chance, we do the same as FIFO, but consider the second-chance index. If the page is found in the cache, we set that page’s index to found. If not, we set every page with index 1 to 0, and replace the first 0 index page with the new page. We also wrap around when the last element is traversed.

1. Provide a complete write-up that will include a test of your solutions and a comparison of **the hit rates (1-fault rate)** for the different algorithms you have implemented. Create a table and plot a graph of the results you got from running your code with the accesses.txt file to represent your findings.

I ran the tests on the accesses.txt results for different cache sizes. Here are the results:

(Number is 1 higher than expected because I have an extra print() for the total page faults)



1. Include the output results of running step 6 on your own testInput.txt file, you don’t need to convert these results into graphs and tables though.

