Programming Assignment 5 Report

Andres Hinojosa

Submitted Files:

- **pagetable.h and .cpp:** defines the PageTable class and the page entries array which hold the frame number and valid bit
- **replacement.h and .cpp:** The replacement class which serves as the base for all three algorithms. Its only function that is used is access_page where touch_page, load_page, and replace_page are called.
- **fifo_replacement.h and .cpp** A subclass of replacement, where the FIFO algorithm is used. The constructor, destructor, and load_page and replace_page are implemented.
- **lifo_replacement.h and .cpp** A subclass of replacement, where the LIFO algorithm is used. The constructor, destructor, and load_page and replace_page are implemented.
- **lru_replacement.h and .cpp** A subclass of replacement, where the LRU algorithm is used. The constructor, destructor, load_page, replace_page, and touch_page as LRU is the only algorithm that moves a page when accessed and it is already in the algorithm.
- **main.h and .cpp** The main program which acts as the driver for the tests. Small refs and large refs text files are opened here and simulated. A time clock was added to measure the time it took to run each algorithm.

How to compile and run the program:

- To compile the program, use command "make."
- To run the program, use command: ./prog5 followed by two integer parameters which are the page size in bytes, and the physical memory in MB respectively.

Results for programs:

- After testing the program with a 1024-page size and 32 MB physical memory which is what I believe was used for the auto grader, I was able to see the amount of time it took for each algorithm to run. Each one managed to be under one second. The results are shown below:

FIFO	LIFO	LRU
0.07 seconds	0.10 seconds	0.90 seconds

Features implemented:

- **Access_page:** The first feature that I implemented was the access_page function in the replacement implementation file. I did this by creating a variable called "pages." This variable holds the number of pages that are available and decreases each time a new page is added to the array. When this number reaches zero, replace_page is called but it is not implemented here since it will be overridden. The same goes for touch_page although that function only has a purpose in the LRU algorithm.

- **FIFO**: The second feature I implemented was the FIFO algorithm. It was the first algorithm I implemented, and it was simple to create. The data structure that I chose was a queue. This is because a queue follows the FIFO algorithm, so it made sense. Each time a page is added, it is added to the back. When a page needs to be replaced, the first element which is also the first to arrive is popped.
- **LIFO**: The next feature that I implemented was the LIFO algorithm. This algorithm was the easiest to create. This is because the most recently added page is the one to be removed. A vector can be used to implement this since its push and pop functions do just that. My LIFO implementation was like my FIFO, so some functions had to be replaced.
- **LRU:** The final feature that I implemented was the LRU algorithm. It was the most difficult to create. I started off by going with my FIFO approach since an LRU is basically FIFO except for the case where a page is touched. In doing this, I had slow run times in the touch_page function since I was searching through the entire array. In the end, I figured out a solution by replacing the queue with a doubly linked list and having it work the exact same way. From there, I added a hashmap for direct access to the nodes in the middle. This way access_page was instant in finding and removing a node instead of having to search through each node.

Design and implementation choices:

This assignment was challenging for me to start, as I was not too sure about the page table size and memory size. I realized it wasn't too important for what I'd be implementing. I was easily able to implement the first two algorithms by following the lecture but came at a stop when it was time to implement the LRU. I first decided to implement a queue as my data structure for it, but my runtimes were too slow. After hearing about a student's advice on how we should use a doubly linked list, I decided to replace my queue with that. I still had the same exact run times since I was using it the exact same way. Finally, I decided to use a hash map that was mentioned in lecture since I believed that would fix the main problem, I was having in touch_page. In having instant access to nodes, I would not have to iterate through the entire list making my program run as fast as fifo and lifo.

Lessons Learned:

In this assignment, a lesson that I re learned that it is always best to start early. I have started each programming assignment early which greatly helped me but decided to focus on other things this time around since finals is coming up and having many due dates. Although I started my program later, I was able to finish it before the due date, but it was a bit risky. If I had started earlier, I would also have gone to office hours to seek some help. This is because I was stuck on the LRU for a while in search of a data structure that would lead to a run time of less than a second.

References:

The reference I used were mostly the lecture slides. I also used notes from CS311 to find the correct data structures for fifo and lifo. I used the website called cplusplus.com in order to learn about the functions supported by the data structures I wanted to use and to make sure I used them correctly. Finally, I used the GeeksforGeeks website as it had some information of how hash maps work and how I could implement it to my doubly linked list.

Future improvements:

To improve my program, I would remove repetitive code that was found in the constructors of each of my algorithms. I would put the appropriate variables in the replacement implementation file to make it neater. Other than that, I would try to reduce the amount of code by figuring out how to make things shorter and more concise.