**CS 4310 Operating Systems**

**Project #2 Simulating Page Replacement Manager and Performance Analysis**

**Due: 11/21**

(Total: 100 points)

**Student Name: \_\_\_\_\_Branden Hitt\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date: \_\_\_\_\_\_\_\_\_11/20\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

***Important:   
-*** *Please read this document completely before you start coding.   
- Also, please read the submission instructions (provided at the end of this document) carefully before submitting the project.*

***Project #2 Description:***

Simulating Page Replacement Manager of the Operating Systems by programming the following three page replacement algorithms that we covered in the class:

1. First In First Out (FIFO)
2. Least Recently Used (LRU)
3. Optimal Algorithm

You can use either Java, or C++ for the implementation. The objective of this project is to help student understand how above three page replacement algorithms operates by implementing the algorithms, and conducting a performance analysis of them based on the performance measure of ***page faults*** for each page replacement algorithm using multiple inputs. Output the details of each algorithm’s execution. You need to show what pages are inside the page frames along with the reference string and mark it when a page fault occurred. You can choose your display format, for examples, you can display the results for each reference string in a table format as shown in the class notes. The project will be divided into three phases to help you to accomplish above tasks in in a systematic and scientific fashion: Design and Testing, Implementation, and Performance Analysis.

The program will read in a reference string a file (ReferenceString.txt) – this file will be generated by you. In this project, assume that

1. the length of the reference string is always 30, e.g. 361724720354720146353214567012
2. there are 8 pages, from 0 to 7.

Note that you need to generate 50 testing cases, or 50 reference strings of length 30. You are required to run each reference string using three algorithms with *NumberOfPageFrame* (3, 4, 5 and 6) page frames.

A sample input file of having 5 page frames and a reference string 361724720354720146353214567012 (always with length 30) is given as follows:

[Begin of ReferenceString.txt]

*NumberOfPageFrame* value:   
5

Reference String:

361724720354720146353214567012

[End of ReferenceString.txt]

You can implement the algorithms in your choice of data structures based on the program language of your choice. Note that you always try your best to give the most efficient program for each problem.

***Submission Instructions:***

* ***turn in the following @blackboard.cpp.edu after the completion of all three parts, part 1, part 2 and part 3***
  + 1. ***(1) three program files (your choice of programming language with proper   
        documentation)***

1. ***this document (complete all the answers)***

**Part 1**

**Design & Testing (30 points)**

* 1. Design the program by providing pseudocode or flowchart for each page replacement algorithm.

First In First Out

Create the page frame (Queue)

Create a counter for the page faults

For each reference in the reference string

Keep track of current page reference

if the frame is not yet full

if the page is already in the frame

do nothing

else

add the page to the frame and increment the counter

else (frame is full)

if the page is already in the frame

do nothing

else

pop off the top of the page frame queue and push a page

increment

Return the count of page faults

Least Recently Used

Create the page frame (Queue)

Create a counter for the page faults

For each reference in the reference string

Keep track of the current page frame reference

if the frame is not yet full

if the page is already in the frame

push the node (page ref) to the back of the queue

else add the page to the frame and increment the counter

else (frame is full)

if the page is already in the frame

push the node (page ref) to the back of the queue

else

pop of the top of the page frame queue and push a page

increment the counter

Return the count of page faults

Optimal Algorithm

Create the page frame (queue)

Create a counter for the page faults

For each page reference in the reference string

Keep track of the current page reference

Keep track of the lookahead (the remaining slice of reference string)

If the page frame is not yet full

If the page is already in the frame

Push that node (page ref) to the back of the queue

Else

Add a page to the frame and increment the counter

Else (page frame is full)

If the page is already in the frame

Push that node (page ref) to the back of the queue

Else

Find the optimal page in the queue to remove based on the lookahead

Remove the optimal page and push the current page

Increment the fault counter

Return the count of page faults

* 1. Design the program correctness testing cases. Give at least 4 testing cases (with 3, 4, 5, or 6, page frames) to test your program, and give the expected correct output (# of page faults) of the program for each case in order to test the correctness of each algorithm.

<complete the following table>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Testing case # | Input  (Input file with a number of page frames value and a reference string) | Expected # of page faults for FIFO  (√ if Correct after testing in Part ) | Expected # of page faults for LRU  (√ if Correct after testing in Part 3) | Expected # of page faults for Optimal Algorithm (√ if Correct after testing in Part 3) |
| 1  (3 page frames) | *NumberOfPageFrame* value:  3  Reference String:  534674043017320060373412473146 | 23.0  √ | 23.0  √ | 17.0  √ |
| 2 (4 page frames) | *NumberOfPageFrame* value:  4  Reference String:  534674043017320060373412473146 | 18.0  √ | 20.0  √ | 14.0  √ |
| 3  (5 page frames) | *NumberOfPageFrame* value:  5  Reference String:  534674043017320060373412473146 | 14.0  √ | 13.0  √ | 11.0  √ |
| 4  (6 page frames) | *NumberOfPageFrame* value:  6  Reference String:  534674043017320060373412473146 | 11.0  √ | 13.0  √ | 10.0  √ |

* 1. Design testing strategy for the programs. Discuss about how to generate and structure the randomly generated inputs for experimental study later in Part 3.

*Hint 1: To study the performance evaluation of the three page replacement algorithms, it is the easiest to use a random number generator for generating the inputs. However, student should store each data set of 50 trails (reference strings) for each of the four frame sizes (3, 4, 5, 6) and use the same data set for running each of the three page replacement algorithms.*

*Hint 2: The average performance result (number of page faults) of each input data size can be calculated after an experiment is conducted in 50 trails (reference strings) of each page frame size (3, 4, 5, 6). We can denote the run time results (number of page faults) as the set X = {x1, x2, x3 … x50} which contains 50 results (number of page faults) for each page frame size, and each xi is the number of page faults of each trail (reference string number i).  
  
The Average time =*

My first approach to testing these algorithms was to make sure that they performed correctly. I went through a few small reference strings by hand and found what my result should be. From there I tested each algorithm to make sure that its results matched up with my own.

Once I had confirmed that my algorithms were performing correctly, I began to randomly generate reference strings that I could use to test and compare the different algorithms at different sizes. For instance, I took one reference string and recorded results of all 3 algorithms at the page frame size 3. I then took the same reference string and tested the algorithms again at page frame sizes of 4, 5, and 6.

This was a good starting point to see how the page fault amounts of each algorithm went down as the page frame size grew. However, I was still unsure as to whether or not my results would behave the same for a different reference string.

Following the first successful trial, I then modified my program to be able to generate a reference string for each trial and to run through the process of all three algorithms being tested at the four page frame sizes. At 50 trials, I now had enough results to average them all out individually and to find more accurate behaviors of the algorithms competing against each other at the four page frame sizes.

I should also note my design for creating the reference strings. Each string was made up randomly of 30 numbers that represented the 8 pages in my simulation (0-7). For example: 534674043017320060373412473146. The reference string was then passed to all the algorithms and then stored in a file (referenceStrings.txt) after use.

**Part 2**

**Implementation (30 points)**

1. Code each program based on the design (pseudocode or flow chart) in Part 1(a).

<generate three programs and stored them in three files, needed to be submitted>

1. Document the program appropriately.

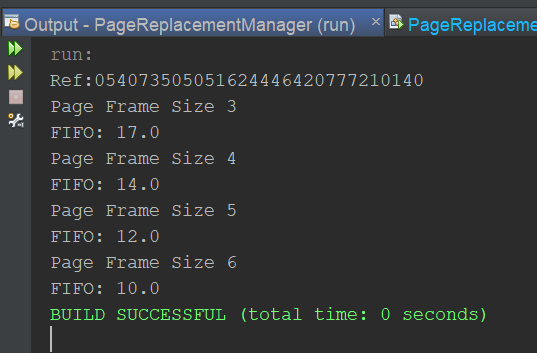
<generate documentation inside the three program files>

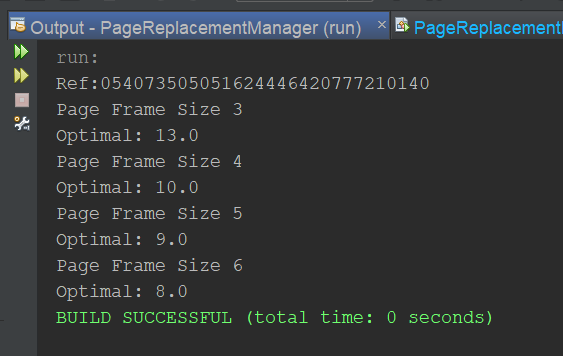
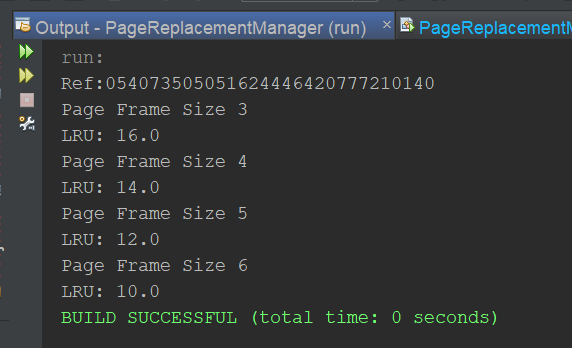
1. Test you program using the designed testing input data given in the table in Part 1(b), Make sure each program generates the correct answer by marking a “√” if it is correct for each testing case for each program column in the table. Repeat the process of debugging if necessary.

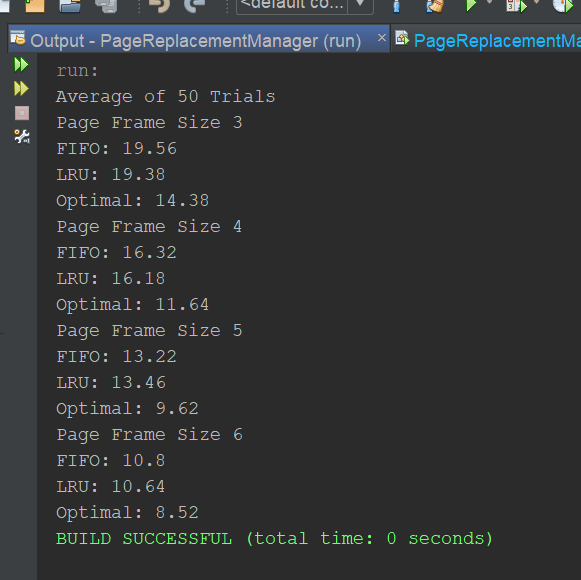
<Complete the three columns of the three algorithms in the table @Part 1(b)>

1. For each program, capture a screen shot of the execution (Compile&Run) using one testing case to show how this program works properly

<Insert totally three screen shots, one for each program, here>







(One example of 50 trials would be output like this)

By now, three working programs are created and ready for experimental study in the next part, Part 3.

**Part 3   
Performance Analysis (100 points)**

1. Run each program with the designed randomly generated input data given in Part 1(c). Generate a table for all the experimental results for performance analysis as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| number of page frames | Average of the total completion time  (FIFO Program) | Average of the total completion time  (LRU Program) | Average of the total completion time  (Optimal Algorithm) |
| 3 page frames | 21,19,20,15,21,21,19,22,22,21  20,19,21,22,20,17,22,17,22,21  20,18,15,21,23,24,19,19,22,22  17,18,24,19,19,21,18,23,24,22  23,18,17,13,21,19,17,21,14,19  AVG = 19.76 | *19,22,21,18,17,19,25,19,16,20*  *17,18,18,20,22,20,23,18,24,14*  *19,22,23,20,13,21,15,20,17,17*  *24,21,17,17,19,20,15,21,20,23*  *21,21,20,21,19,22,17,23,23,14*  AVG = 19.68 | *11,15,16,11,15,19,15,15,18,16*  *14,16,14,15,11,15,15,17,11,14*  *13,15,17,15,18,13,12,12,15,17*  *15,16,16,14,16,12,16,14,14,15*  *14,14,16,15,16,15,12,14,15,16*  *AVG = 14.64* |
| 4 page frames | *18,15,20,17,17,13,17,20,20,20*  *18,14,15,20,19,14,15,18,14,20*  *19,17,14,17,17,14,17,20,17,19*  *18,16,17,14,12,18,17,16,17,19*  *18,17,17,17,13,16,16,17,14,15*  *AVG = 16.32* | *18,15,14,17,19,19,16,17,15,14*  *17,19,17,10,16,22,16,20,18,21*  *13,13,17,21,14,17,19,17,14,19*  *18,15,17,14,17,15,12,18,19,17*  *17,13,15,16,18,16,16,17,20,19*  *AVG = 16.26* | *10,13,14,13,12,15,12,11,13,12,13*  *11,14,14,13,11,12,12,13,11,13*  *12,10,12,13,14,11,11,11,13,10*  *15,8,13,11,12,10,12,11,12,11*  *10,13,13,12,13,10,13,11,12*  *AVG = 11.92* |
| 5 page frames | *13,13,15,11,13,16,15,9,13,13*  *10,17,12,11,15,17,16,15,15,13*  *15,12,18,12,12,16,8,13,13,16*  *9,17,16,17,11,17,13,10,15,18*  *16,15,14,14,9,9,11,12,10,10*  *AVG = 13.46* | *15,11,11,12,12,16,12,10,15,13*  *16,13,17,14,14,15,11,15,8,14*  *14,13,14,11,11,13,11,14,17,17*  *12,12,16,14,12,16,9,10,12,10*  *18,17,15,16,14,13,12,16,15,19*  *AVG = 13.68* | *9,10,12,10,11,9,11,11,8,10*  *11,9,10,10,10,10,9,10,10,11*  *9,8,11,11,9,9,10,9,9,9*  *7,11,10,10,10,10,11,9,9,9*  *10,12,10,12,10,9,11,11,8,10*  *AVG = 10.10* |
| 6 page frames | *13,10,11,12,12,9,13,8,12,12*  *11,10,12,8,7,13,13,9,14,11*  *12,7,10,13,8,8,12,13,12,13*  *8,11,13,10,11,9,10,11,14,9*  *11,8,12,7,12,11,12,9,11,14*  *AVG = 10.92* | *12,13,10,10,9,15,11,15,11,14*  *11,11,7,11,10,11,13,11,12,8*  *10,7,15,10,9,12,13,8,10,12*  *11,9,7,10,9,13,8,11,10,10*  *9,12,10,11,14,10,13,11,9,8*  *AVG = 10.94* | *9,10,10,8,9,8,9,8,9,8*  *9,9,7,8,8,9,10,9,11,8*  *9,9,9,9,9,8,8,9,9,8*  *9,10,10,11,9,9,9,8,8,11*  *7,9,8,8,9,9,9,7,9,10*  *AVG = 8.76* |

1. Plot a graph of each algorithm, average page fault vs. page frame size (3, 4, 5, 6) and summarize the performance of each algorithm based on its own graph.

FIFO was a good baseline to start out with. At a page frame size of 3, there were many faults. As the page frame size grew larger, the number of faults fell.

Surprisingly, the LRU graph looked very similar to the FIFO graph. Although it seemed like the approach of removing the least-recently-used page may be useful, it did not have any effect. The results were almost the exact same as the FIFO results. This seems to me to be due to the fact that the page calls were random, meaning that any page could be needed regardless of how often it is used. Perhaps in an environment that kept track of common processes, this sort of approach could prove to be more useful. Still, in this current simulation, the LRU algorithm did not help.

The optimal algorithm was exactly as it boasted. At the starting page frame size of 3, the average amount of page faults was around 14. Compared to the other two algorithms, that is 5 less page faults on average at that page frame size.

The obvious problem with this algorithm is that the lookahead does not actually exist, meaning that this algorithm is not feasible in real world scenarios. Still, I think that looking ahead could somehow translate into a similar pseudo-optimal version of this system. Or at the very least, there could be specific scenarios that show similar patterns that could be improved similarly to the way the optimal algorithm improved the paging system in this simulation.

Plot all three graphs on the same graph and compare the performance (page faults) of all three algorithms. Rank three page replacement algorithms. Try giving the reasons for the findings.  

As you can see from the above graphs, the FIFO and LRU performed almost exactly the same. Their plot lines are difficult to distinguish from each other due to the extremely similar results. This leads me to believe that implementing least recently used is no more effective than just removing by a first in first out method.

On the other hand, you can also see from the results that the optimal algorithm is just as it says: optimal. Regardless of the page frame size, the optimal algorithm performed better than the other two.

Its important to note that, for all three algorithms, they had less page faults as the page frame size grew. This makes sense as more frames could be stored in the frame at a time and there would be less of a chance of needing to swap one out for another. If the page frame size were to equal the number of pages being references, there would only be a fault for the first time each page entered the frame. However, implementing a page frame of that size would be very costly and unrealistic.

1. Conclude your report with the strength and constraints of your work. At least 100 words.

(Note: It is reflection of this project. If you have a change to re-do this project again, what you like to keep and what you like to do differently in order get a better quality of results.)

For this project, I decided to code in JAVA and to use a similar data structure for all three algorithms. The strength of my implementation of the algorithms was that I used a modified queue to handle the page frame. This allowed me to easily pop off the front (FIFO) or to push a node backwards (my main modification used for LRU). Using a queue allowed me to visualize the problem better and to be efficient in handling certain insertion operations.

My weakness in this project was searching for specific nodes in the queue. I found out after implementation and testing that I could incorporate a hash map to better handle searching for the correct node rather than traversing the queue. The hash map could have held the addresses of the nodes inside the queue which could have provided me a search time of O(1).

If I were to continue this problem, I would rewrite the functionality with a hash map and maybe compare the different languages to see which ran the fastest.