**CS 4310 Operating Systems**

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

**Due: 11/21**

(Total: 100 points)

**Student Name: Bryce Callender**

**Date: 11/4/2019**

***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.

FIFO

Init the number of page faults to 0

Init index to start at to 0

Make a containter to hold the info with a size of the page frame

For each char in the referenceString

Get the page number by subtracting by ascii 0 since char is a different number

If the number is not in page buffer

Place the pagenumber at the index

Add 1 to page fault counter

Add one to index and mod it by the size of page frame

Return the page fault counter

LRU

Init the number of page faults to 0

Make a containter to store the pages with an initial value to show nothing in the pos

Make a containter to store the recent pages accessed

Keep track of placement index and set to 0

For each char in the referenceString

Get the page number value

If the number is not in the page buffer

If the page frame is not full

Add it

Increment placement index

Else

Get the front of the recently used vector and remove it

Place the new page number in that position that was deleted

Push the number to the LRU vector

Increment the page fault counter

Else

If the number is found

Delete it

Re-add the number to the back simulating it being recently used

Return page fault count

Optimal

Init the number of page faults to 0

Make a containter to store the pages with initial value of -1

Keep track of placement index and set to 0

For each char in the reference string

Get the page number value

If the number is not in the page frame

If the page frame is not full

Add the page to the vector

Increment index

Else

Make a unorderered map to keep track of used items

Go through page frame and set the values to 1

Make variable j and set to iterate over the remaining values in the reference string

Cache a var for the current page number

While we are not past the length and there’s not one page number with a 1 in the map

change page number to current position of the reference string at index j’s page number

increment j

if the new page number is in the map

if that map value is > 0

decrement the value

make variable to store the page to remove

for each entry in the map

if the value of the number is 1 that is the page to remove

find the position of that number

set the page to that cached current page number above the while

increment page fault counter

return page fault count

* 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:  361724720354720146353214567012 | 27  √ | 27  √ | 20  √ |
| 2 (4 page frames) | *NumberOfPageFrame* value:  4  Reference String:  361724720354720146353214567012 | 25  √ | 27  √ | 16  √ |
| 3  (5 page frames) | *NumberOfPageFrame* value:  5  Reference String:  361724720354720146353214567012 | 23  √ | 25  √ | 14  √ |
| 4  (6 page frames) | *NumberOfPageFrame* value:  6  Reference String:  361724720354720146353214567012 | 15  √ | 17  √ | 12  √ |

* 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 =*

For testing page replacement, I will be doing 50 trials of each algorithm and taking the average page fault count. There will be the same reference string for each algorithm, however a random number generator will build a new string for each new test for the algorithms. The reference strings are of length 30 and only 30. The page numbers are from the range of 0 to 7. The page frame sizes will be 3,4,5, and 6.

**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>

<DONE>

1. Document the program appropriately.

<generate documentation inside the three program files>  
<DONE>

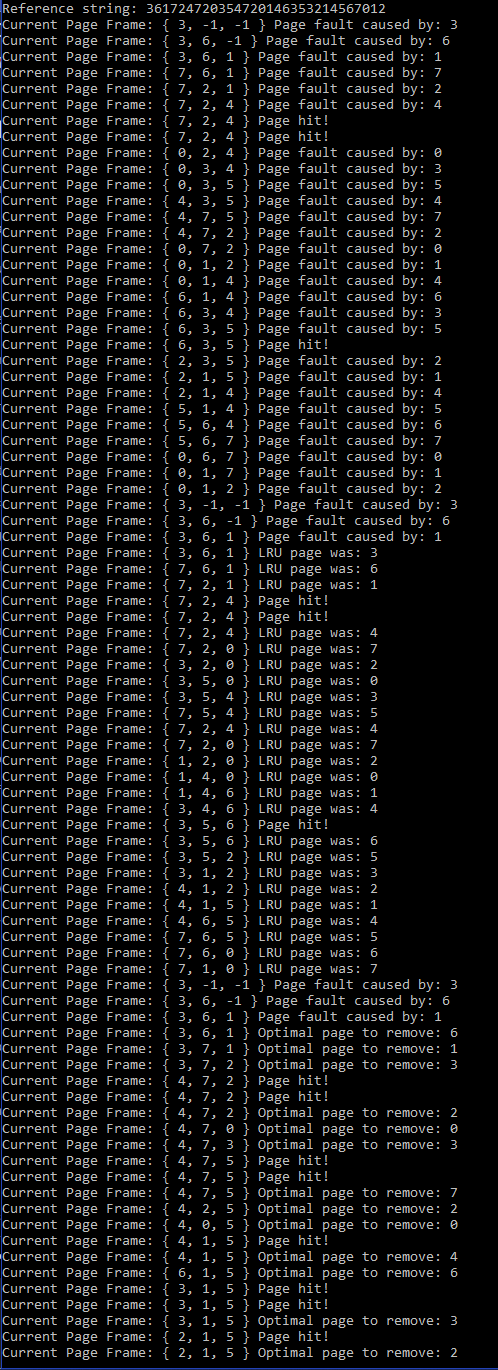
1. Test you program using the designed testing input data given in the table in Part 1(b), Make sure each program generates the correct answer by marking a “√” if it is correct for each testing case for each program column in the table. Repeat the process of debugging if necessary.

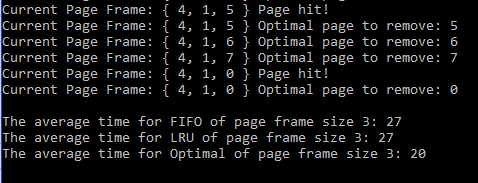
<Complete the three columns of the three algorithms in the table @Part 1(b)>

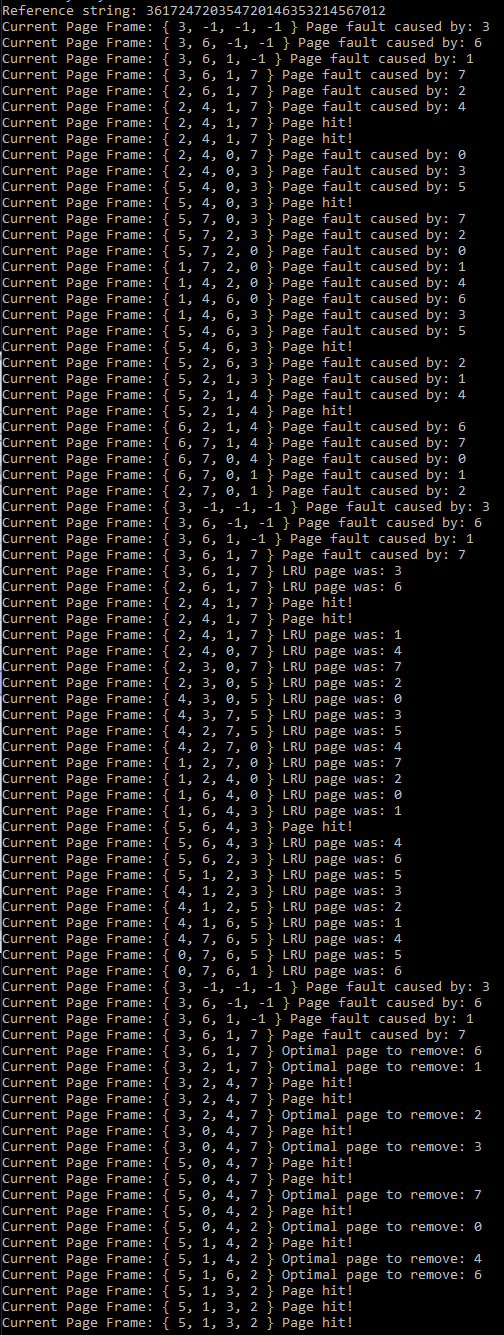
<DONE>

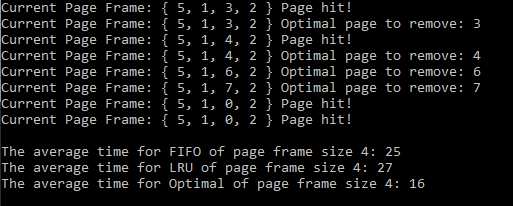
1. For each program, capture a screen shot of the execution (Compile&Run) using one testing case to show how this program works properly

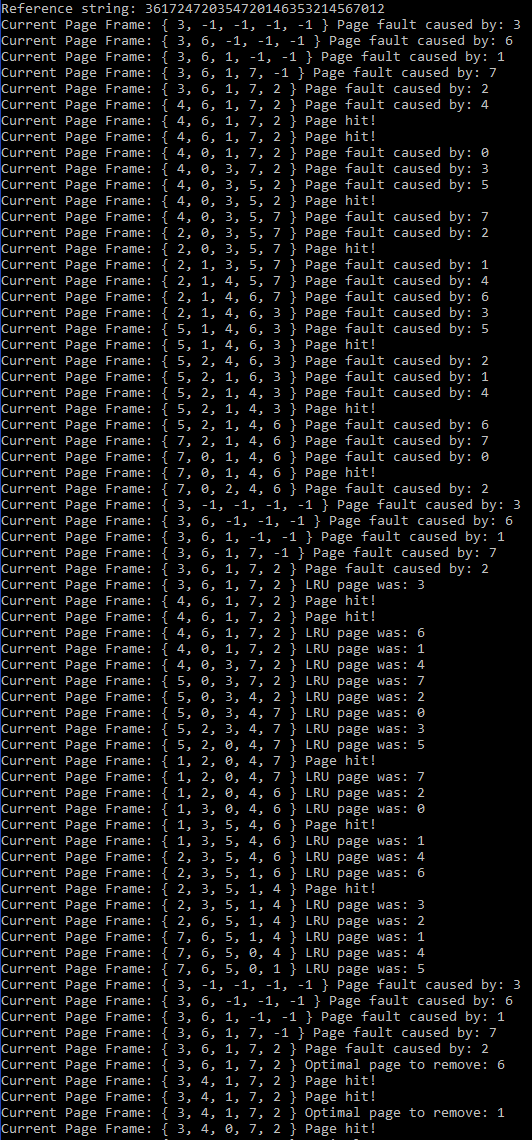
<Insert totally three screen shots, one for each program, here>

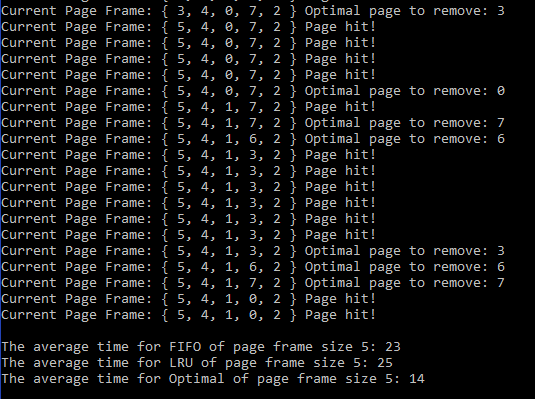


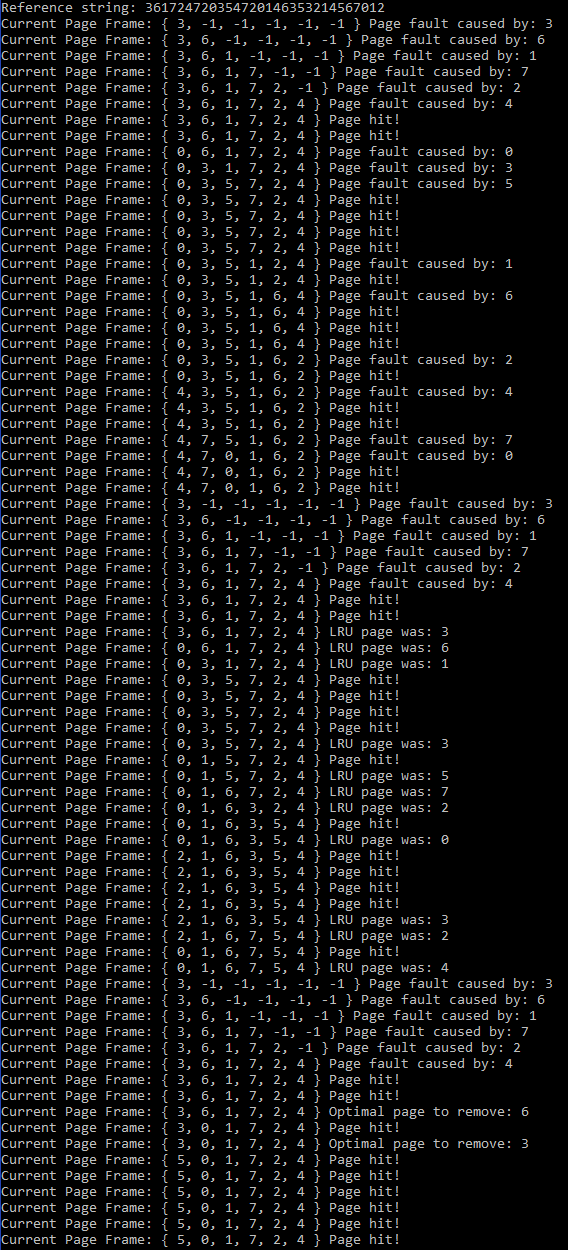


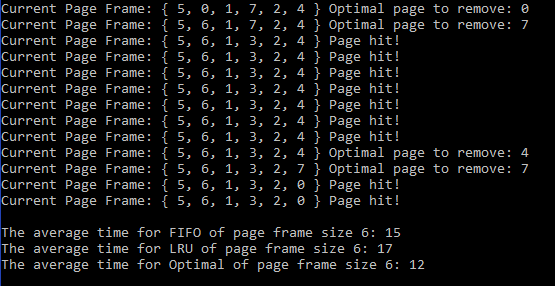












By now, three working programs are created and ready for experimental study in the next part, Part 3.

**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 | 19.04 | 19.42 | 14.3 |
| 4 page frames | 16.28 | 16.42 | 11.74 |
| 5 page frames | 14.22 | 14.2 | 10 |
| 6 page frames | 11.02 | 11.14 | 8.54 |

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.

<Insert totally three graphs, one for each program, here>

FIFO algorithm was okay it’s not the best, however it’s the easiest to use. When the page frame increases it gets a smaller number of faults, but not by that much.

LRU seemed to do okay as well. This algorithm had less faults as the page frame size increased. LRU would seem to be a good algorithm however this data depends heavily on the string it receives.

Optimal algorithm responded well to each reference string size. Performed the best with the smallest of the page frames. It didn’t go above 15 with page frame of 3 which is great.

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.  

This graph ranks them as Optimal, FIFO, and then LRU. The Optimal algorithm will always be the best as it can investigate the future to figure out what is going to occur the latest so it will get rid of that. The thing with FIFO and LRU is that they are dependent on the reference string. For this the reference strings with FIFO seemed to perform better than the LRU. This could mean the LRU got rid of something it shouldn’t have like if the number is immediately following the current position. The LRU algorithm could have beaten out FIFO if the strings were different since it relies heavily on the random number generator to make strings its good at.

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.)

I feel like this project was a success. This project was interesting as we got to be the computer in how it keeps track of caching pages. My reference string random generator was a uniform int distribution so the strings should have a uniform distribution of numbers. However, looking at the file there was a lot of duplicate numbers. If I did the program again, I would try to minimize the number of occurrences to really test the algorithm and how it handles these faults. The duplicates are good to showcase the page hits. I would also like to test more with FIFO and LRU. They were always so close with the reference strings my random number generated. If I did this again, I would like to see cases where the average time is a heavily different number.