CSN-232
Operating Systems
LRU Cache Implementation Report

Group Members

Ankita Bansal	15112011
Ashutosh Parija	15116010
Himanshu Gupta	15112042
Sakshi Goyal	15116046
Satyam Yadav	15116052
Surya Narayan	15116059
Utkarsh Gupta	15116066
Vipul Gupta	15116073

Codes: Github link

Introduction

Cache replacement algorithms/policies are optimizing instructions that a computer program can utilize in order to manage a cache of information stored on the computer.
Least Recently Used (LRU) page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the pages that are used very less already are likely to be used less in the future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. A page fault occurs when a referenced page in not found in the memory frames.

The following algorithms have been analyzed and implemented:

- Counter Method
- Stack Method
- Aging Method
- Clock Method

Test Cases

Please find the test cases here.

We have used a input of 50 integers some of which consists of repeating characters and some consists of non-repeating characters.

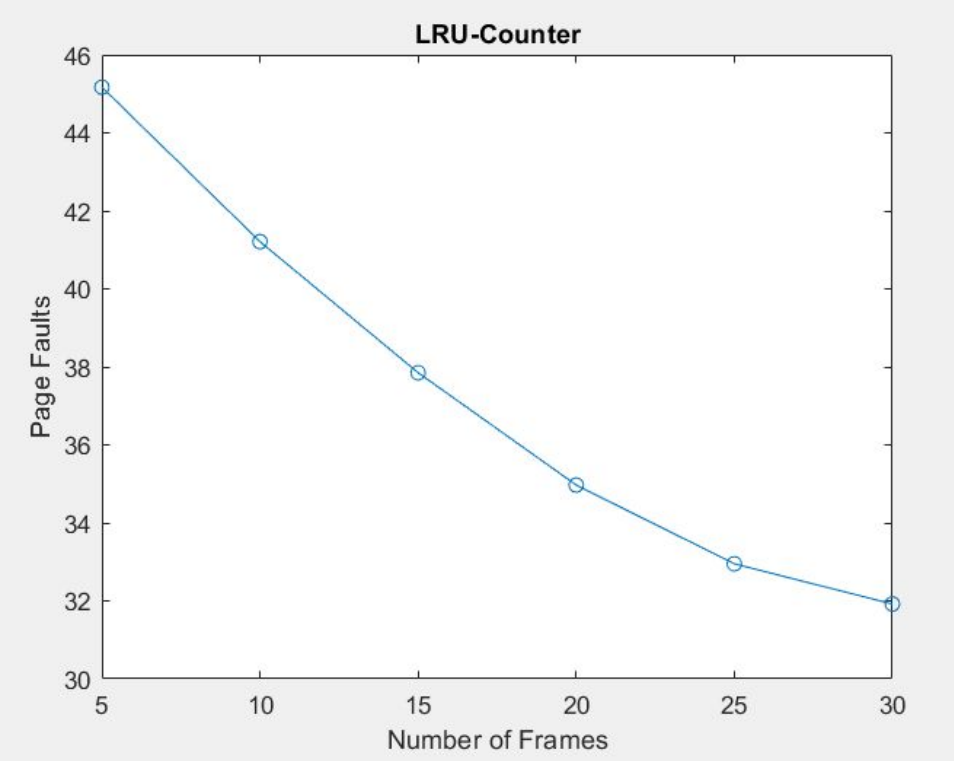
E.g.
35 16 48 50 3 50 16 6 2 44 36 6 3 31 12 6 24 42 11 37 10 32 29 49 2 47 34 23 31 39 9 2 30 47 34 50 17 47 15 39 1 16 33 28 45 35 13 47 41 44

Also, we have taken a sample case consisting of 10 values i.e. 3, 1, 4, 2, 5, 2, 1, 2, 3, 4 and cache size of 4 in the codes implemented.

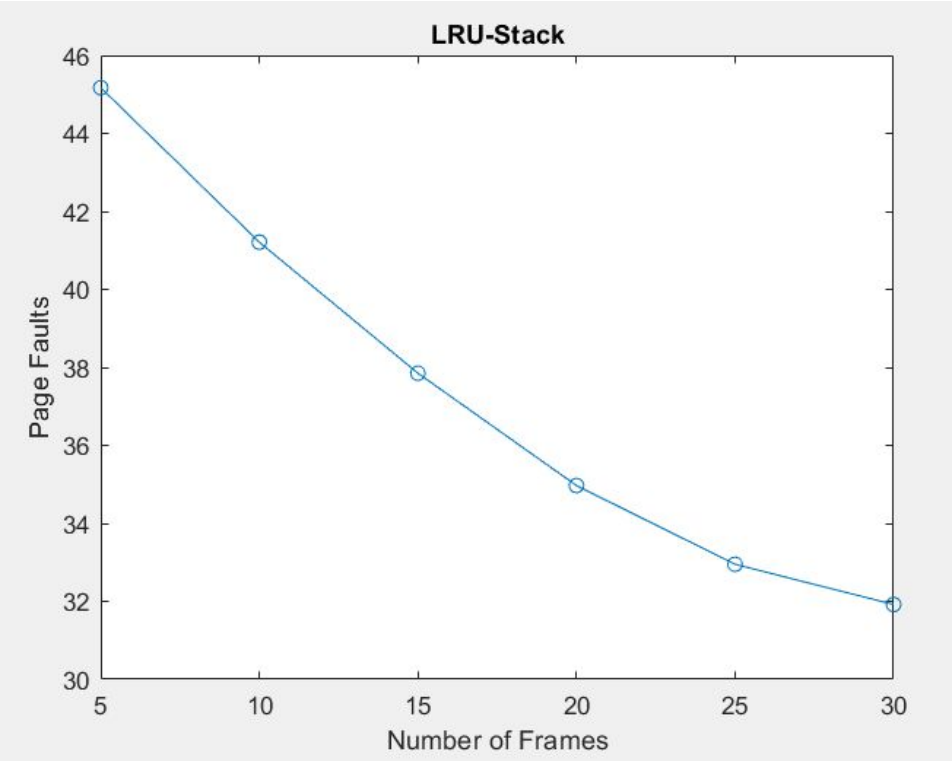
Graphical Results

(Number of Page Faults vs Number of frames)
The following results are interpreted with an input size of 50.

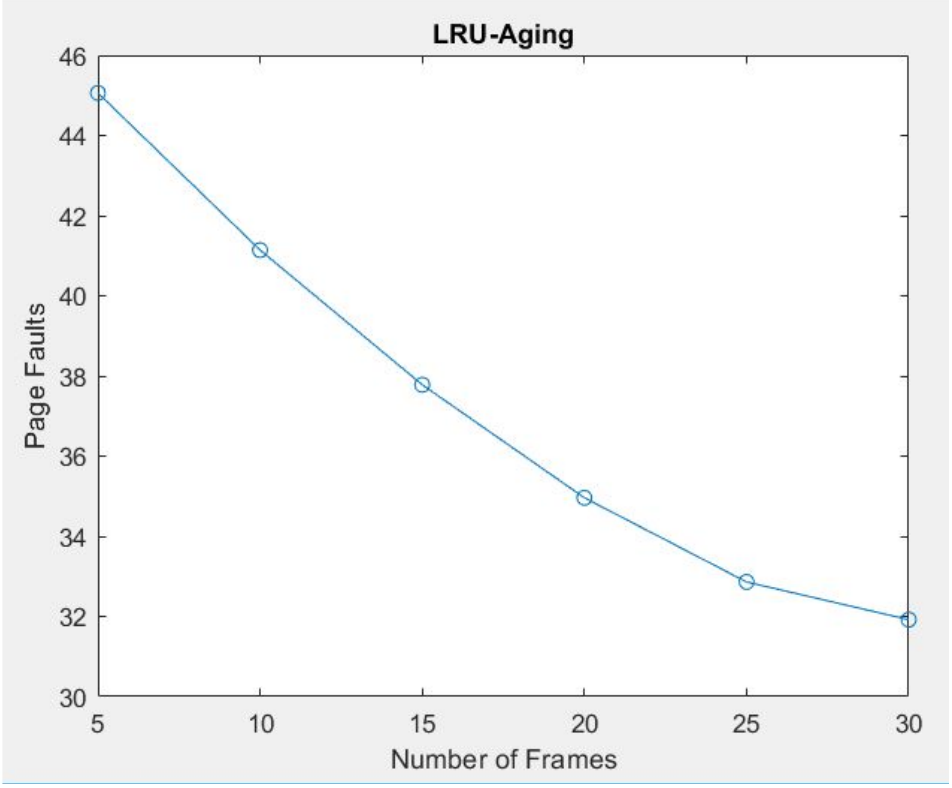
1. *Counter Implementation*



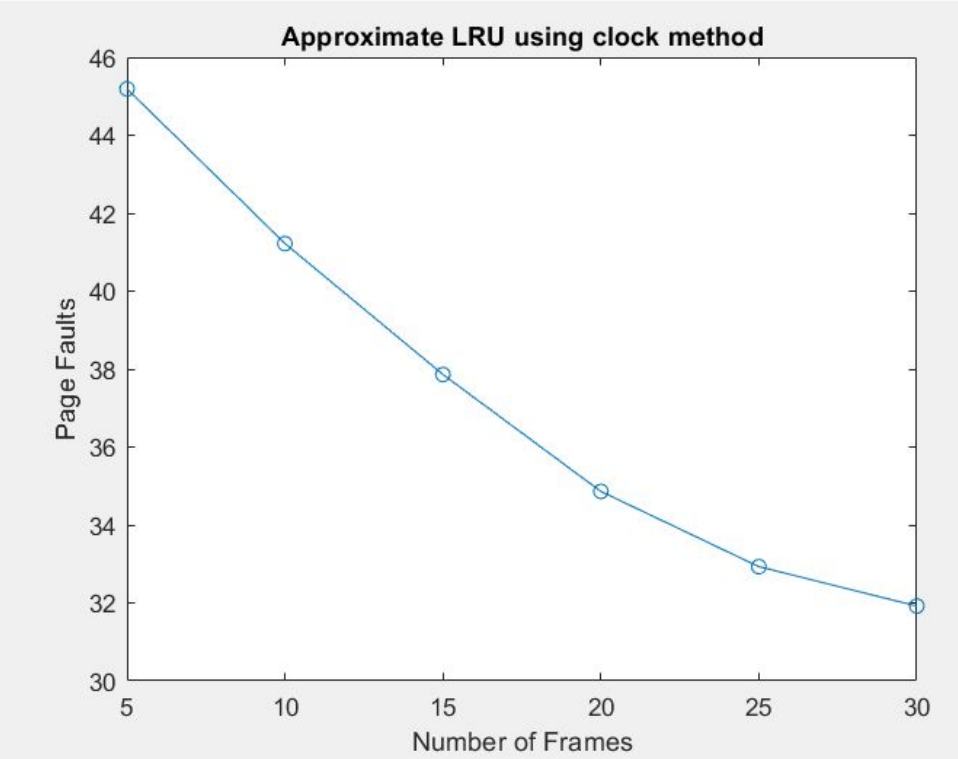
2. *Stack Implementation*



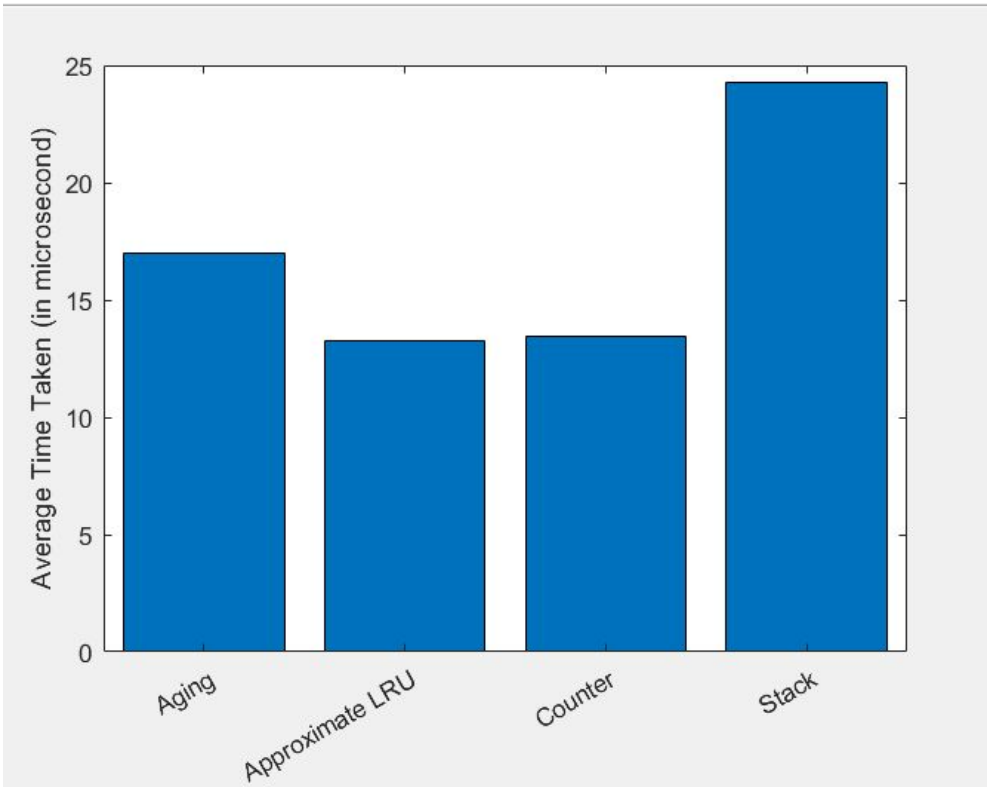
3. *Aging Register Implementation*



4. *Clock Implementation*



Average Execution Time for different Corner Cases



Complexity Analysis

Having implemented the LRU algorithms, now we calculate the running time complexities of the same.

1. *Counter Implementation*

   Here as we see according to the codes, the function checkOverFlow() implemented runs only when now approaches 2^30.

   Therefore safely assuming that the aforementioned function doesn't run in average cases, we calculate the time complexity of the algorithm.

   Assuming the worst case and no. of pages > no. of frames available, we get the time complexity as:

   $T(a, c) = O(a*c - c^2)$

   where a = No. of accesses/ Input size
   c = No. of frames/ Capacity of cache

   Keeping the no. of accesses constant, we see that time taken reduces as the capacity of the cache increases.

   Moreover, if the checkOverflow function runs ever, the time complexity increases by an order of capacity.

2. *Stack Implementation*

   Here, we find the time complexity to be:

   $T(a, p) = O(a*p)$

   where a = No. of accesses/ Input size
   p = No. of pages

3. *Aging Implementation*

   Here, the time complexity depends mainly upon the access counter function. reset_page_counter() would add a constant multiplier to the time.
   Thus,

   $T(a,c) \propto a*(a-1)*c$

   where a = No. of accesses/ Input size
   c = No. of frames/ Capacity of cache

or $T(a,c) = O(c*a^2)$

4. Clock Implementation

Here, we analyse the time complexities for individual functions as:

secondChance(): $T(a, c) = O(a*c)$
clockPRA(): $T(a, c) = O(a*c)$
optPRA: $T(a, c) = O(a*c^2)$

where a = No. of accesses/ Input size
c = No. of frames/ Capacity of cache

Therefore, the overall worst case complexity is: $T(a, c) = O(a*c^2)$