OS-LAB 9-(CSLR42) - 20/04/2021

Rajneesh Pandey - 106119100

Page Replacement Algorithms | (FIFO)

Example . Consider page reference string 1, 3, 0, 3, 5, 6 and 3 page slots.

Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 **Page Faults**.
when 3 comes, it is already in memory so —> 0 Page Faults.
Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 **Page Fault**.
Finally 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 **Page Fault**.

So total page faults = 5.

Implementation – Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1- Start traversing the pages.
i) If set holds less pages than capacity.
a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
b) Simultaneously maintain the pages in the queue to perform FIFO.
c) Increment page fault
ii) Else
If current page is present in set, do nothing.
Else
a) Remove the first page from the queue as it was the first to be entered in the memory
b) Replace the first page in the queue with the current page in the string.
c) Store current page in the queue.
d) Increment page faults.

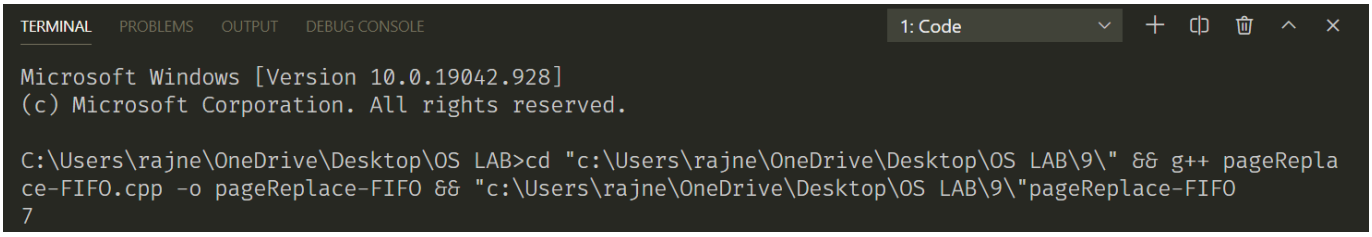
2. Return page faults.

9 > C++ PageReplace-FIFO.cpp > pageFaults(int [], int, int)

```cpp
//106119100 Rajneesh Pandey

#include <bits/stdc++.h>
using namespace std;
int pageFaults(int pages[], int n, int capacity)
{
    unordered_set<int> s;
    queue<int> indexes;
    int page_faults = 0;
    for (int i = 0; i < n; i++)
    {
        if (s.size() < capacity)
        {
            if (s.find(pages[i]) == s.end())
            {
                s.insert(pages[i]);
                page_faults++;
                indexes.push(pages[i]);
            }
        }
        else
        {
            if (s.find(pages[i]) == s.end())
            {
                int val = indexes.front();
                indexes.pop();
                s.erase(val);
                s.insert(pages[i]);
                indexes.push(pages[i]);
                page_faults++;
            }
        }
    }
    return page_faults;
}
int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = 4;
    cout << pageFaults(pages, n, capacity);
    return 0;
}
```

Input / Output



Page Replacement Algorithms | (LRU)

Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1- Start traversing the pages.
i) If set holds less pages than capacity.
a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
b) Simultaneously maintain the recent occurred index of each page in a map called indexes.
c) Increment page fault
ii) Else
If current page is present in set, do nothing.
Else
a) Find the page in the set that was least recently used. We find it using index array. We basically need to replace the page with minimum index.
b) Replace the found page with current page.
c) Increment page faults.
d) Update index of current page.

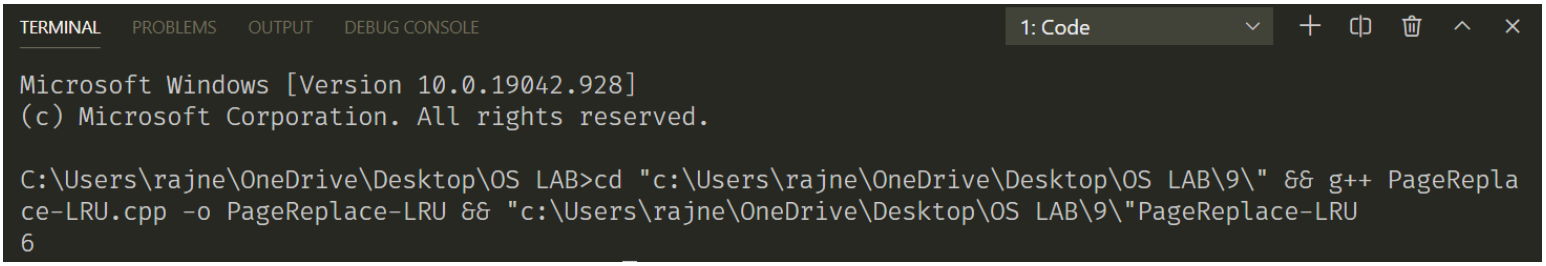
2. Return page faults.

9 > C++ PageReplace-LRU.cpp > ...

```cpp
//106119100 Rajneesh Pandey

#include <bits/stdc++.h>
using namespace std;

int pageFaults(int pages[], int n, int capacity)
{
    unordered_set<int> s;
    unordered_map<int, int> indexes;
    int page_faults = 0;
    for (int i = 0; i < n; i++)
    {
        if (s.size() < capacity)
        {
            if (s.find(pages[i]) == s.end())
            {
                s.insert(pages[i]);
                page_faults++;
            }
            indexes[pages[i]] = i;
        }
        else
        {
            if (s.find(pages[i]) == s.end())
            {
                int lru = INT_MAX, val;
                for (auto it = s.begin(); it != s.end(); it++)
                {
                    if (indexes[*it] < lru)
                    {
                        lru = indexes[*it];
                        val = *it;
                    }
                }
                s.erase(val);
                s.insert(pages[i]);
                page_faults++;
            }
            indexes[pages[i]] = i;
        }
    }
    return page_faults;
}
int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = 4;
    cout << pageFaults(pages, n, capacity);
    return 0;
}
```



Banker's Algorithm in Operating System

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Why Banker's algorithm is named so?
Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

Following Data structures are used to implement the Banker's Algorithm:

Let 'n' be the number of processes in the system and 'm' be the number of resources types.

Available :

It is a 1-d array of size 'm' indicating the number of available resources of each type.
Available[j] = k means there are 'k' instances of resource type Rj
Max :

It is a 2-d array of size 'n*m' that defines the maximum demand of each process in a system.
Max[i, j] = k means process Pi may request at most 'k' instances of resource type Rj.
Allocation :

It is a 2-d array of size 'n*m' that defines the number of resources of each type currently allocated to each process.
Allocation[i, j] = k means process Pi is currently allocated 'k' instances of resource type Rj
Need :

It is a 2-d array of size 'n*m' that indicates the remaining resource need of each process.
Need [i, j] = k means process Pi currently need 'k' instances of resource type Rj
for its execution.

Need [i, j] = Max [i, j] – Allocation [i, j]

9 > BankersAlgo.cpp > ...

```cpp
// 106119100 Rajneesh Pandey

#include <iostream>
using namespace std;

int main()
{
    int n, m, i, j, k;
    n = 5;
    m = 3;
    int alloc[5][3] = {{0, 1, 0},
                       {2, 0, 0},
                       {3, 0, 2},
                       {2, 1, 1},
                       {0, 0, 2}};

    int max[5][3] = {{7, 5, 3},
                     {3, 2, 2},
                     {9, 0, 2},
                     {2, 2, 2},
                     {4, 3, 3}};

    int avail[3] = {3, 3, 2};

    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++)
    {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    int y = 0;
    for (k = 0; k < 5; k++)
    {
        for (i = 0; i < n; i++)
        {
            if (f[i] == 0)
            {

                int flag = 0;
                for (j = 0; j < m; j++)
                {
                    if (need[i][j] > avail[j])
                    {
                        flag = 1;
                        break;
                    }
                }

                if (flag == 0)
                {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }
```

```
63
64      cout << "Following is the SAFE Sequence" << endl;
65      for (i = 0; i < n - 1; i++)
66          cout << " P" << ans[i] << " →";
67      cout << " P" << ans[n - 1] << endl;
68
69      return (0);
70  }
```

Input / Output

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE                    1: Code

Microsoft Windows [Version 10.0.19042.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\rajne\OneDrive\Desktop\OS LAB>cd "c:\Users\rajne\OneDrive\Desktop\OS LAB\9\" && g++ BankersAl
go.cpp -o BankersAlgo && "c:\Users\rajne\OneDrive\Desktop\OS LAB\9\"BankersAlgo
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```