**Page Replacement Algorithms**

**Experiment No: 6 Date: 04/10/23**

**Aim:** **a)** To implement FIFO (first in first out)Page Replacement Algorithm.

**Theory:**

The First-In, First-Out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little bookkeeping on the part of the operating system12. The operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the oldest arrival in front. When a page needs to be replaced, the page in the front of the queue is selected for removal

The FIFO page replacement algorithm follows these steps:

Step 1: Initialize a queue to keep track of the pages in memory.

Step 2: When a page fault occurs and a page needs to be replaced, the page at the front of the queue is selected for replacement.

Step 3: The page that is being replaced is removed from memory and the new page is added to the back of the queue.

Step 4: This process continues each time a page fault occurs and a page needs to be replaced

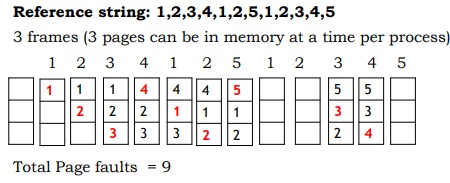
Advantages

* Simple to implement:
* Low overhead:
* Easy to understand
* Predictable
* Predictable behavior:
* Low complexity:
* Suitable for workloads with temporal locality

Disadvantages

* Belady’s AnomalyNot
* effective for workloads with a spatial locality

Example



**CODE:**

#include <iostream>

using namespace std;

int main() {

int f, n;

cout << "\nEnter the number of frames: ";

cin >> f;

cout << "\nEnter number of page references: ";

cin >> n;

int data[n], alloc[f];

// Initialize alloc array with blank spaces

for (int i = 0; i < f; i++)

alloc[i] = -1; // Initialize with -1

cout << "\nEnter Page String: ";

for (int i = 0; i < n; i++)

cin >> data[i];

cout << endl;

int ind = 0, fault = 0;

for (int i = 0; i < n; i++) {

int flag = 0;

for (int j = 0; j < f; j++) {

if (alloc[j] == data[i])

flag = 1;

}

if (flag != 1) {

alloc[ind] = data[i];

ind = (ind + 1) % f;

fault++;

}

cout<<"Current State: ";

for (int k = 0; k < f; k++) {

if (alloc[k] != -1)

cout << alloc[k] << " ";

else

cout.width(4); // set the width of the next field

cout.fill(' '); // fill character for the next field

}

cout << endl;

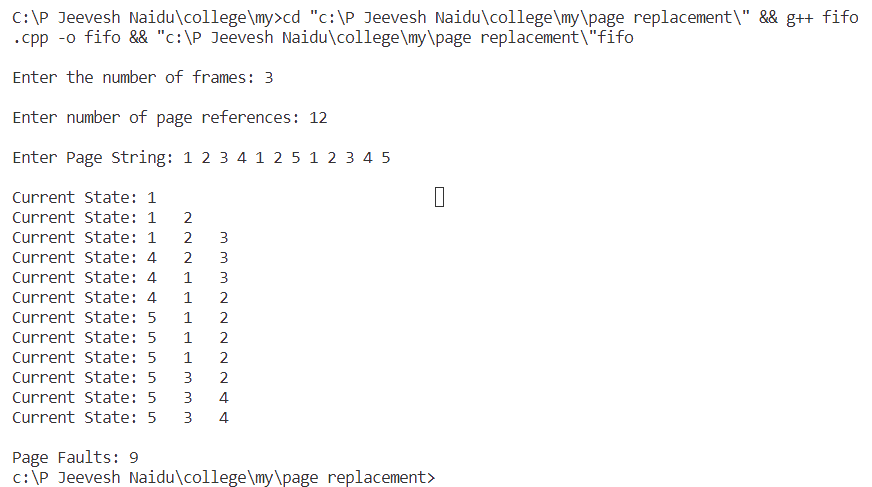
}

cout << "\nPage Faults: " << fault;

return 0;

}

**OUTPUT:**

****

**Conclusion:**

FIFO (first come first out) Page Replacement Algorithm was successfully implemented in this experiment

**Page Replacement Algorithms**

**Experiment No: 6 Date: 11/10/23**

**Aim:** **b)** To implement Optimal Page Replacement Algorithm.

**Theory:**

The optimal page replacement algorithm is a memory management technique that minimizes the number of page faults by predicting future accesses and replacing the least recently used pages12. It uses a "stack-based algorithm" for replacing pages2. The algorithm pushes pages into the stack as per the memory demand until the memory is full, and then pops the least recently used page when the queue is full3. The optimal page replacement algorithm achieves the minimum number of page faults in theory

First, it checks if the requested page is already present in the main memory. If so, it detects a page hit and proceeds to the next page request.

If the page is not found in the main memory, it triggers a page miss and checks if the main memory is full.

When the main memory is not full, it loads the requested page into an empty slot and moves on to the next page request.

If the main memory is full, it first makes room for the new page. To do so, it finds the page that will not be accessed for the longest period in the future. After that, it replaces the selected page with the newly requested page. This ensures that the main memory remains occupied by the most relevant and frequently accessed pages

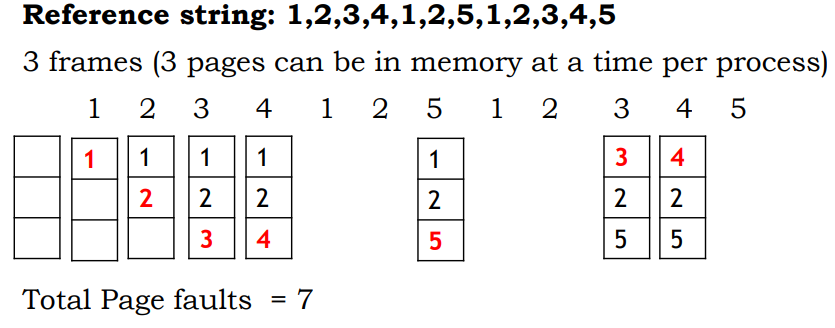
Advantages

* t is easy to implement.
* · It requires minimum fuss is to replace the page.

Disadvantages

* It is very difficult to detect the errors present in the algorithm
* It is not support in all types of "operating system"

Example



**CODE:**

#include<iostream>

#include<string.h>

using namespace std;

int faults = 0;

int return\_index(int \*arr, int n, int search);

int ifpresent(int n, int \*arr, int value, int start);

int last\_page(int n, int \*arr, int \*frames, int curr, int fn);

int main() {

int n, ch, page\_no, flag, j = 0, replace\_index, nop, frames\_empty, lp, lp\_index;

cout << "\nEnter the number of frames: ";

cin >> n;

frames\_empty = n;

int frames[n], replace[n], y = 0;

char c[1];

for (int i = 0; i < n; i++) {

replace[i] = frames[i] = -1; // Initialize with -1

}

cout << "\nEnter number of page references: ";

cin >> nop;

int pages[nop];

cout << "\nEnter Page String: ";

for (int i = 0; i < nop; i++) {

cin >> pages[i];

}

cout << endl;

for (int y = 0; y < nop; y++) {

page\_no = pages[y];

for (int i = 0; i < n; i++) {

if (frames[i] != page\_no) {

flag = 1;

} else {

flag = 0;

break;

}

}

if (flag == 1) {

if (frames\_empty > 0) {

frames[n - frames\_empty] = page\_no;

} else {

lp = last\_page(nop, pages, frames, y, n);

if (lp != -1) {

lp\_index = return\_index(frames, n, lp);

} else {

lp\_index = 0;

}

frames[lp\_index] = page\_no;

}

frames\_empty--;

faults++;

} else {

}

cout<<"Current State: ";

for (int i = 0; i < n; i++) {

if (frames[i] != -1)

cout << frames[i] << " ";

else

cout.width(4); // set the width of the next field

cout.fill(' '); // fill character for the next field

}

cout << endl;

}

cout << "\nTotal Page faults = " << faults;

return 0;

}

int return\_index(int \*arr, int n, int search) {

for (int i = 0; i < n; i++) {

if (arr[i] == search) {

return i;

}

}

return -1;

}

int ifpresent(int n, int \*arr, int value, int start) {

for (int i = start; i < n; i++) {

if (arr[i] == value) {

return 1;

}

}

return 0;

}

int last\_page(int n, int \*arr, int \*frames, int curr, int fn) {

int got[fn], k = 0;

for (int i = 0; i < fn; i++) {

if (!ifpresent(n, arr, frames[i], curr + 1)) {

return frames[i];

}

}

int last = -1;

for (int i = curr + 1; i < n; i++) {

for (int j = 0; j < fn; j++) {

if (arr[i] == frames[j] && !ifpresent(k, got, arr[i], 0)) {

last = arr[i];

got[k++] = last;

}

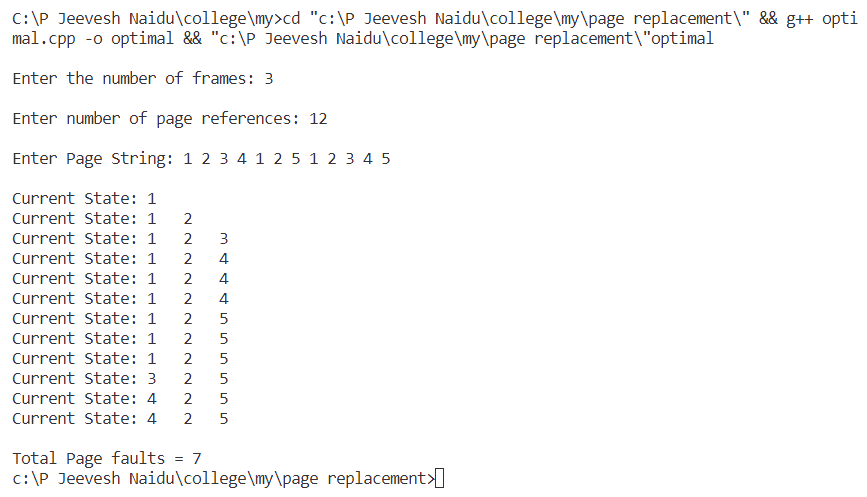
}

}

return got[--k];

}

**OUTPUT:**

****

**Conclusion:**

Optimal Page Replacement Algorithm was successfully implemented in this experiment

**Page Replacement Algorithms**

**Experiment No: 6 Date: 18/10/23**

**Aim:** **c)** To implement LRU (Least Recently Used) Page Replacement Algorithm.

**Theory:**

As studied in Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more processes into memory at the same time. but what happens when a process requests for more pages and no free memory is available to bring them in. Following steps can be taken to deal with this problem:

1. Put the process in the wait queue, until any other process finishes its execution thereby freeing frames.
2. Or, remove some other process completely from the memory to free frames.
3. Or, find some pages that are not being used right now, move them to the disk to get free frames. This technique is called Page replacement and is most commonly used.

In Virtual Memory Management, Page Replacement Algorithms play an important role. The main objective of all the Page replacement policies is to decrease the maximum number of page faults.

Page Fault – It is basically a memory error, and it occurs when the current programs attempt to access the memory page for mapping into virtual address space, but it is unable to load into the physical memory then this is referred to as Page fault.

This algorithm helps to decide which pages must be swapped out from the main memory in order to create a room for the incoming page. This Algorithm wants the lowest page-fault rate.  
LRU algorithm stands for "Least recent used" and this algorithm helps the Operating system to search those pages that are used over a short duration of time frame.

* The page that has not been used for the longest time in the main memory will be selected for replacement.
* This algorithm is easy to implement.
* This algorithm makes use of the counter along with the even-page

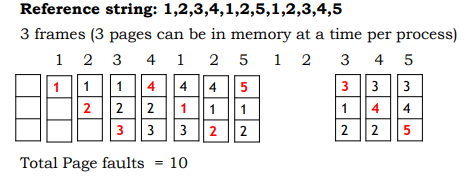
Advantages of LRU

* It is an efficient technique.
* With this algorithm, it becomes easy to identify the faulty pages that are not needed for a long time.
* It helps in Full analysis.

Disadvantages of LRU

* It is expensive and has more complexity.
* There is a need for an additional data structure.

Example



**CODE:**

#include<bits/stdc++.h>

#include<iostream>

using namespace std;

int LRU(int time[], int f) {

int i, minimum = time[0], pos = 0;

for (i = 1; i < f; i++) {

if (time[i] < minimum) {

minimum = time[i];

pos = i;

}

}

return pos;

}

int main() {

int nf, np, alloc[20], pages[30], counter = 0, time[20];

int flag1, flag2, i, j, pos, faults = 0;

cout << "Enter number of Frames: ";

cin >> nf;

cout << "Enter number of pages: ";

cin >> np;

cout << "Enter pages: ";

for (i = 0; i < np; i++)

cin >> pages[i];

for (i = 0; i < nf; i++)

alloc[i] = -1;

cout << "\n";

for (i = 0; i < nf; i++) {

cout << "F" << i + 1 << " ";

}

cout << "\n";

for (i = 0; i < np; i++) {

flag1 = flag2 = 0;

for (j = 0; j < nf; j++) {

if (alloc[j] == pages[i]) {

counter++;

time[j] = counter;

flag1 = flag2 = 1;

break;

}

}

if (flag1 == 0) {

for (j = 0; j < nf; j++) {

if (alloc[j] == -1) {

counter++;

faults++;

alloc[j] = pages[i];

time[j] = counter;

flag2 = 1;

break;

}

}

}

if (flag2 == 0) {

pos = LRU(time, nf);

counter++;

faults++;

alloc[pos] = pages[i];

time[pos] = counter;

}

cout << "\n";

cout << "Current State: ";

for (j = 0; j < nf; j++) {

if (alloc[j] != -1) {

cout << alloc[j] << " ";

} else {

cout << " "; // Print a blank space

}

}

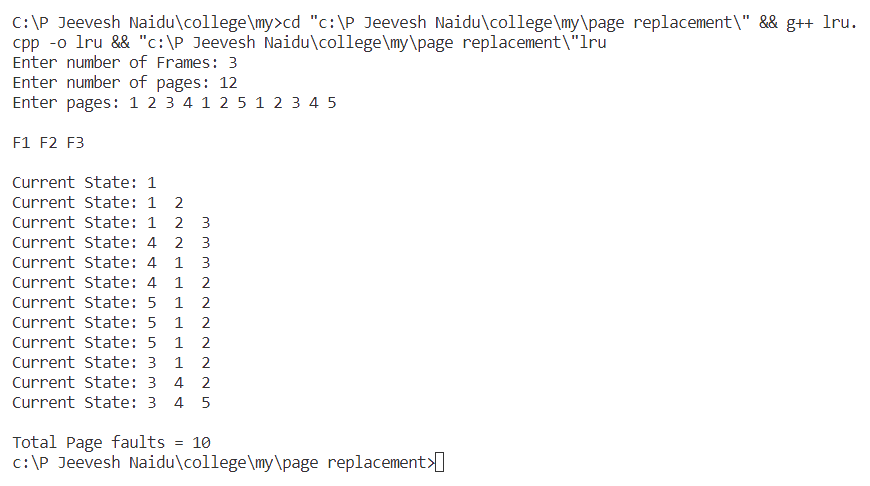
}

cout << "\n\nTotal Page faults = " << faults;

return 0;

}

**OUTPUT:**

****

**Conclusion:**

LRU (Least Recently Used) Page Replacement Algorithm was successfully implemented in this experiment.