Experiment No.9
Memory Management: Virtual Memory a Write a program in C demonstrate the concept of page replacement policies for handling page faults eg: FIFO, LRU, Optimal
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Memory Management: Virtual Memory

Objective:

To study and implement page replacement policy FIFO, LRU, OPTIMAL

Theory:

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.

Page Replacement Algorithm

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated.

Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference.

First In First Out (FIFO)

This is the simplest page replacement algorithm. In this algorithm, the OS maintains a queue that keeps track of all the pages in memory, with the oldest page at the front and the most recent page at the back.

When there is a need for page replacement, the FIFO algorithm, swaps out the page at the front of the queue, that is the page which has been in the memory for the longest time.

Least Recently Used (LRU)

Least Recently Used page replacement algorithm keeps track of page usage over a short period of time. It works on the idea that the pages that have been most heavily used in the past are most likely to be used heavily in the future too.

In LRU, whenever page replacement happens, the page which has not been used for the longest amount of time is replaced.

Optimal Page Replacement

Optimal Page Replacement algorithm is the best page replacement algorithm as it gives the least number of page faults. It is also known as OPT, clairvoyant replacement algorithm, or Belady's optimal page replacement policy.

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future, i.e., the pages in the memory which are going to be referred farthest in the future are replaced.

This algorithm was introduced long back and is difficult to implement because it requires future knowledge of the program behavior. However, it is possible to implement optimal page replacement on the second run by using the page reference information collected on the first run.

Program:

```
#include <stdio.h>
```

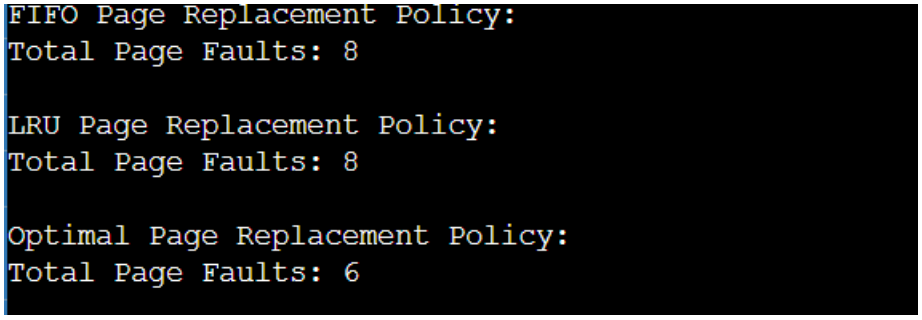
```
#include <stdlib.h>
#include <stdbool.h>
#define NUM_FRAMES 3
#define NUM_PAGES 10
// Function prototypes
void FIFO(int pages[], int n);
void LRU(int pages[], int n);
void optimal(int pages[], int n);
int main() {
    int pages[NUM_PAGES] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3};
    printf("FIFO Page Replacement Policy:\n");
    FIFO(pages, NUM_PAGES);
    printf("\nLRU Page Replacement Policy:\n");
    LRU(pages, NUM_PAGES);
    printf("\nOptimal Page Replacement Policy:\n");
    optimal(pages, NUM_PAGES);
    return 0;
}
void FIFO(int pages[], int n) {
    int frame[NUM_FRAMES] = {-1};
    int page_faults = 0;
    int frame_index = 0;

    for (int i = 0; i < n; i++) {
        bool page_found = false;
        for (int j = 0; j < NUM_FRAMES; j++) {
            if (frame[j] == pages[i]) {
                page_found = true;
                break;
            }
        }
        if (!page_found) {
            frame[frame_index] = pages[i];
            frame_index = (frame_index + 1) % NUM_FRAMES;
```

```
            page_faults++;
        }
    }

    printf("Total Page Faults: %d\n", page_faults);
}

void LRU(int pages[], int n) {
    int frame[NUM_FRAMES] = {-1};
    int page_faults = 0;
    int counter[NUM_FRAMES] = {0}; // Counter to track the usage of each page in frame

    for (int i = 0; i < n; i++) {
        bool page_found = false;
        int min_counter = counter[0]; // Initialize with the first counter value
        int min_index = 0;

        // Check if the page is already in the frame
        for (int j = 0; j < NUM_FRAMES; j++) {
            if (frame[j] == pages[i]) {
                page_found = true;
                counter[j] = i; // Update the counter for the page
                break;
            }
        }

        // If page not found, find the least recently used page
        if (!page_found) {
            page_faults++;
            for (int j = 0; j < NUM_FRAMES; j++) {
                if (counter[j] < min_counter) {
                    min_counter = counter[j];
                    min_index = j;
                }
```

```
            }
            // Replace the least recently used page with the current page
            frame[min_index] = pages[i];
            counter[min_index] = i; // Update the counter for the new page
        }
    }

    printf("Total Page Faults: %d\n", page_faults);
}

void optimal(int pages[], int n) {
    int frame[NUM_FRAMES] = {-1};
    int page_faults = 0;

    for (int i = 0; i < n; i++) {
        bool page_found = false;
        for (int j = 0; j < NUM_FRAMES; j++) {
            if (frame[j] == pages[i]) {
                page_found = true;
                break;
            }
        }
        if (!page_found) {
            int index_to_replace = -1;
            for (int j = 0; j < NUM_FRAMES; j++) {
                bool found_in_future = false;
                for (int k = i + 1; k < n; k++) {
                    if (frame[j] == pages[k]) {
                        found_in_future = true;
                        break;
                    }
                }
                if (!found_in_future) {
                    index_to_replace = j;
```

```
                break;
            }
        }
        if (index_to_replace == -1) {
            int max_future_index = -1;
            for (int j = 0; j < NUM_FRAMES; j++) {
                int future_index = -1;
                for (int k = i + 1; k < n; k++) {
                    if (frame[j] == pages[k]) {
                        future_index = k;
                        break;
                    }
                }
                if (future_index > max_future_index) {
                    max_future_index = future_index;
                    index_to_replace = j;
                }
            }
        }
        frame[index_to_replace] = pages[i];
        page_faults++;
    }
  }

  printf("Total Page Faults: %d\n", page_faults);
}
```

Output:



Conclusion:

Why do we need page replacement strategies ?

Page replacement strategies are crucial in virtual memory management to optimize memory utilization and ensure efficient handling of memory demands. They enable the operating system to efficiently manage limited physical memory by dynamically swapping pages between main memory and secondary storage. Without effective page replacement strategies, the system may suffer from thrashing, where excessive page swapping occurs, leading to degraded performance and system instability. By intelligently selecting which pages to replace based on various algorithms, page replacement strategies help maintain system responsiveness and prevent excessive disk I/O, ultimately enhancing overall system performance and stability.