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.

Code :

```
#include <stdio.h>

#include <stdlib.h>


#define PAGE_SIZE 4

#define MEMORY_SIZE 16

#define NUM_PAGES MEMORY_SIZE / PAGE_SIZE

#define NUM_FRAMES 4


int main() {

   int page_table[NUM_PAGES] = {-1}; // Initialize page table with invalid frame numbers

   int memory[NUM_FRAMES][PAGE_SIZE]; // Physical memory frames

   int next_frame = 0; // Index to keep track of next available frame

   int page_faults = 0;


   printf("Demand Paging Simulation (FIFO Page Replacement):\n");


   FILE *fp;

   fp = fopen("pages.txt", "r"); // Read page references from file "pages.txt"

   if (fp == NULL) {

      printf("Error opening file.\n");
```

```
    return 1;
}

int page;
while (fscanf(fp, "%d", &page) != EOF) {
    printf("Referencing page %d\n", page);

    // Check if page is already in memory
    int frame = -1;
    for (int i = 0; i < NUM_PAGES; i++) {
        if (page_table[i] == page) {
            frame = i;
            break;
        }
    }

    if (frame == -1) { // Page fault
        page_faults++;
        printf("Page fault occurred.\n");

        // Load page into memory
        if (next_frame < NUM_FRAMES) { // If there are available frames
            frame = next_frame;
            next_frame++;
```

```
        } else { // If all frames are occupied, use FIFO replacement
            frame = 0;
            printf("Using FIFO page replacement policy.\n");

            // Update page table
            page_table[memory[0][0]] = -1; // Evict the first page in memory
        }

        // Update page table and load page into memory
        page_table[frame] = page;
        for (int i = 0; i < PAGE_SIZE; i++) {
            memory[frame][i] = page * PAGE_SIZE + i; // Simulate loading data into
memory
        }
    } else {
        printf("Page hit occurred.\n");
    }
  }

  fclose(fp);

  printf("Total number of page faults: %d\n", page_faults);

  return 0;
}
```

Page Replacement Policies for Handling Page Faults (LRU - Least Recently Used):

Code :

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

#define PAGE_SIZE 4
#define MEMORY_SIZE 16
#define NUM_PAGES MEMORY_SIZE / PAGE_SIZE
#define NUM_FRAMES 4

int main() {
    int page_table[NUM_PAGES] = {-1}; // Initialize page table with invalid frame numbers
    int memory[NUM_FRAMES][PAGE_SIZE]; // Physical memory frames
    int page_access_count[NUM_PAGES] = {0}; // Array to keep track of page access counts
    int page_faults = 0;

    printf("Page Replacement Policy (LRU - Least Recently Used):\n");

    FILE *fp;
    fp = fopen("pages.txt", "r"); // Read page references from file "pages.txt"
```

```
if (fp == NULL) {
    printf("Error opening file.\n");
    return 1;
}

int page;
while (fscanf(fp, "%d", &page) != EOF) {
    printf("Referencing page %d\n", page);

    // Check if page is already in memory
    int frame = -1;
    for (int i = 0; i < NUM_PAGES; i++) {
        if (page_table[i] == page) {
            frame = i;
            break;
        }
    }

    if (frame == -1) { // Page fault
        page_faults++;
        printf("Page fault occurred.\n");

        // Load page into memory
        int min_access_count = page_access_count[0];
```

```
            frame = 0;
            for (int i = 1; i < NUM_PAGES; i++) {
                if (page_access_count[i] < min_access_count) {
                    min_access_count = page_access_count[i];
                    frame = i;
                }
            }

            // Update page table and load page into memory
            page_table[frame] = page;
            for (int i = 0; i < PAGE_SIZE; i++) {
                memory[frame][i] = page * PAGE_SIZE + i; // Simulate loading data into
memory
            }
        } else {
            printf("Page hit occurred.\n");
        }

        // Update page access count
        page_access_count[frame]++;
    }

    fclose(fp);

    printf("Total number of page faults: %d\n", page_faults);
```

```
    return 0;
}
```

Output :

Information about each referenced page, whether it results in a page hit or a page fault.

If a page fault occurs, it would display a message indicating so and mention the chosen replacement policy (FIFO).

At the end, it would display the total number of page faults encountered during the simulation.

For the second code snippet (LRU Page Replacement Policy), the output would be similar:

Information about each referenced page, indicating whether it's a page hit or a page fault.

If a page fault occurs, it would mention it and display the chosen replacement policy (LRU).

At the end, it would show the total number of page faults.

You can run these programs in your local environment by compiling them and providing an input file "pages.txt" containing page references. Then,

Conclution :

In conclusion, the implementation of page replacement policies in C offers a comprehensive understanding of how operating systems manage memory to optimize performance. Through the demonstration of FIFO (First In, First Out), LRU (Least Recently Used), and Optimal page replacement algorithms, we have observed distinct approaches to handling page faults.

FIFO, the simplest of the three, replaces the oldest page in memory, disregarding its recent usage. LRU, on the other hand, prioritizes retaining pages that have been accessed most recently, minimizing the probability of future page faults. Optimal, while theoretically optimal, is not practically implementable but serves as a benchmark for performance evaluation.