LAB Manual

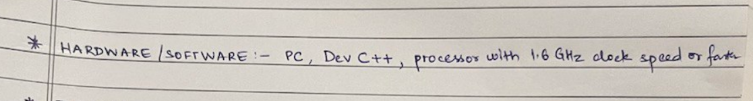
|  |
| --- |
| **Name of Student: Vidhi Binwal PRN: 22070122249** |
| **Semester: IV Year AY 23-24** |
| **Subject Title: Operating Systems Lab** |
| **EXPERIMENT No: 9 Assignment No : 10** |
| **TITLE: Page Replacement Algorithm** **DoP : 23-4-24** |

***Aim:*** Implement C program demonstrate **Page Replacement Algorithm (LRU)**

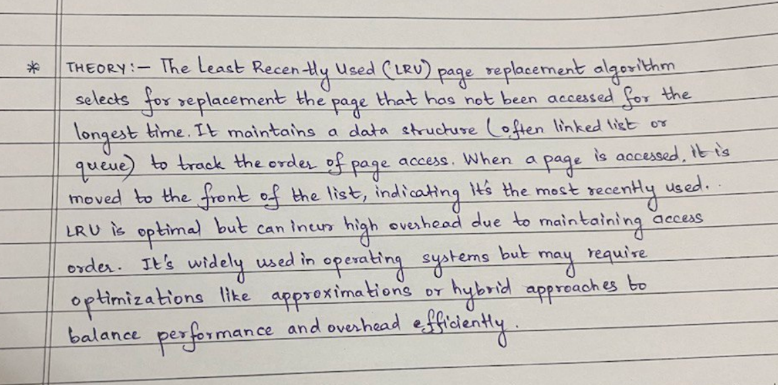
***Learning Outcomes:*** *1. To understand the* **Page Replacement algorithm**

*2. To Demonstrate the working of* **Page Replacement**Shortest algorithm

***Hardware/Software*:**

****

***Theory:***

******

***Algorithm:***

#### Data Structures:

- Maintain a data structure to track the recent usage of pages. A doubly linked list is commonly used for this purpose.

#### Initialization:

1. Initialize an empty doubly linked list to track the pages in memory.

2. Initialize a hash table to map page numbers to their corresponding nodes in the linked list.

#### Accessing a Page:

1. When a page is accessed:

- If the page is already in memory:

- Move the corresponding node to the end of the linked list (indicating it's the most recently used).

- If the page is not in memory (page fault):

- If the memory is not full:

- Create a new node for the page and append it to the end of the linked list.

- Add an entry to the hash table mapping the page number to the new node.

- If the memory is full:

- Remove the node at the front of the linked list (indicating it's the least recently used).

- Remove the corresponding entry from the hash table.

- Create a new node for the new page and append it to the end of the linked list.

- Add an entry to the hash table mapping the page number to the new node.

***Program:***

#include <stdio.h>

#include <stdlib.h>

#define MEMORY\_SIZE 3

// Node structure for the doubly linked list

typedef struct Node {

    int page\_number;

    struct Node\* prev;

    struct Node\* next;

} Node;

// Function to create a new node

Node\* createNode(int page\_number) {

    Node\* newNode = (Node\*)malloc(sizeof(Node));

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(EXIT\_FAILURE);

    }

    newNode->page\_number = page\_number;

    newNode->prev = NULL;

    newNode->next = NULL;

    return newNode;

}

// Function to move a node to the end of the list

void moveToEnd(Node\*\* head\_ref, Node\* node) {

    if (\*head\_ref == NULL || node == NULL)

        return;

    if (\*head\_ref == node)

        \*head\_ref = node->next;

    if (node->next != NULL)

        node->next->prev = node->prev;

    if (node->prev != NULL)

        node->prev->next = node->next;

    while ((\*head\_ref)->next != NULL)

        \*head\_ref = (\*head\_ref)->next;

    (\*head\_ref)->next = node;

    node->prev = \*head\_ref;

    node->next = NULL;

}

// Function to handle page replacement using LRU algorithm

void lruPageReplacement(int page\_reference\_array[], int size) {

    Node\* memory[MEMORY\_SIZE]; // Memory frames

    Node\* page\_list = NULL; // Doubly linked list to track page usage

    int page\_faults = 0;

    // Initialize memory frames to NULL

    for (int i = 0; i < MEMORY\_SIZE; i++)

        memory[i] = NULL;

    // Iterate through page reference array

    for (int i = 0; i < size; i++) {

        int page\_number = page\_reference\_array[i];

        int found = 0;

        // Check if page is already in memory

        for (int j = 0; j < MEMORY\_SIZE; j++) {

            if (memory[j] != NULL && memory[j]->page\_number == page\_number) {

                found = 1;

                moveToEnd(&page\_list, memory[j]);

                break;

            }

        }

        // Page fault

        if (!found) {

            page\_faults++;

            // If memory is full, remove the least recently used page

            if (page\_list != NULL && page\_list->prev != NULL) {

                Node\* temp = page\_list->prev;

                page\_list->prev = temp->prev;

                if (temp->prev != NULL)

                    temp->prev->next = page\_list;

                free(temp);

            }

            // Create a new node for the new page

            Node\* newNode = createNode(page\_number);

            if (page\_list == NULL)

                page\_list = newNode;

            else {

                page\_list->next = newNode;

                newNode->prev = page\_list;

                page\_list = newNode;

            }

            // Add the new page to memory

            memory[MEMORY\_SIZE - 1] = newNode;

        }

    }

    printf("Total Page Faults: %d\n", page\_faults);

}

int main() {

    // Example page reference array

    int page\_reference\_array[] = {1, 3, 0, 3, 5, 6, 3};

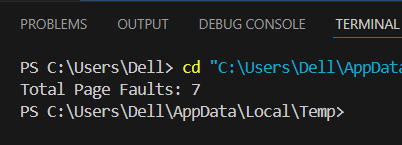
    int size = sizeof(page\_reference\_array) / sizeof(page\_reference\_array[0]);

    lruPageReplacement(page\_reference\_array, size);

    return 0;

}

***Output:***



***Conclusion:***

