

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:

* **HARDWARE / SOFTWARE :-** PC, Dev C++, processor with 1.6 GHz clock speed or faster

Theory:

* **THEORY:-** The Least Recently Used (LRU) page replacement algorithm selects for replacement the page that has not been accessed for the longest time. It maintains a data structure (often linked list or queue) to track the order of page access. When a page is accessed, it is moved to the front of the list, indicating it's the most recently used. LRU is optimal but can incur high overhead due to maintaining access order. It's widely used in operating systems but may require optimizations like approximations or hybrid approaches to balance performance and overhead efficiently.

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;
    }
}

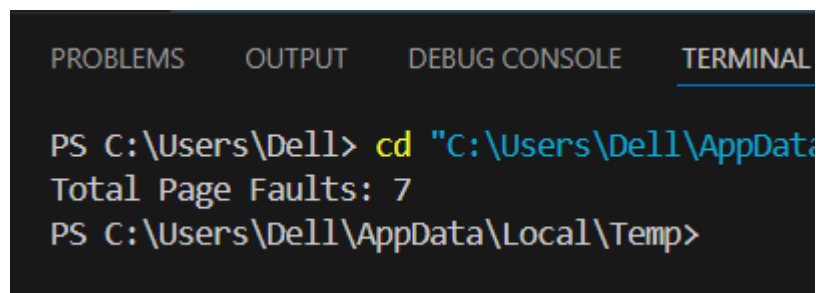
printf("Total Page Faults: %d\n", page_faults);
}

int main() {
    // Example page reference array
    int page_reference_array[] = {1, 3, 0, 3, 5, 6, 3};

```

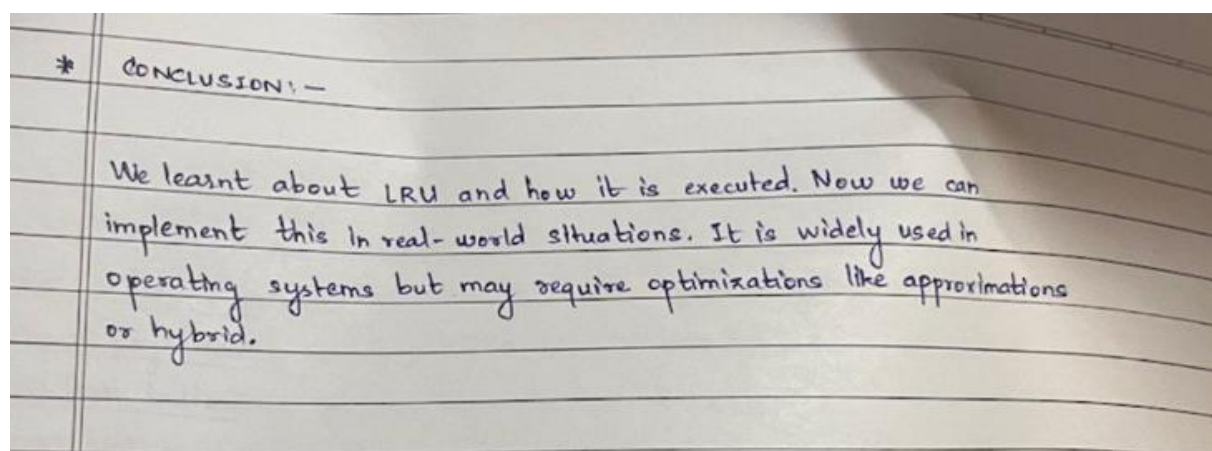
```
int size = sizeof(page_reference_array) / sizeof(page_reference_array[0]);  
  
lruPageReplacement(page_reference_array, size);  
  
return 0;  
}
```

Output:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL  
PS C:\Users\Dell> cd "C:\Users\Dell\AppData\Local\Temp"  
Total Page Faults: 7  
PS C:\Users\Dell\AppData\Local\Temp>
```

Conclusion:



* CONCLUSION:-
We learnt about LRU and how it is executed. Now we can implement this in real-world situations. It is widely used in operating systems but may require optimizations like approximations or hybrid.

