**Task No. 1:** Implement the FIFO and LRU policies described above in C language.

**Solution:**

#include <stdio.h>

#include <stdlib.h>

// FIFO Page Replacement Policy

void fifo(int pages[], int n, int capacity) {

int \*frame = (int \*)malloc(capacity \* sizeof(int));

int frameIndex = 0;

int pageFaults = 0;

int i, j, k, flag;

printf("\nFIFO Page Replacement Policy:\n");

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

flag = 0;

// Check if the page is already in the frame

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

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

flag = 1; // Page found in the frame

break;

}

}

// If page is not found in the frame

if (flag == 0) {

frame[frameIndex] = pages[i];

frameIndex = (frameIndex + 1) % capacity;

pageFaults++;

}

// Print the current state of the frame

printf("Page %d: ", pages[i]);

for (k = 0; k < capacity; k++) {

if (frame[k] == -1)

printf("- ");

else

printf("%d ", frame[k]);

}

printf("\n");

}

printf("\nTotal Page Faults: %d\n", pageFaults);

free(frame);

}

// LRU Page Replacement Policy

void lru(int pages[], int n, int capacity) {

int \*frame = (int \*)malloc(capacity \* sizeof(int));

int \*counter = (int \*)malloc(capacity \* sizeof(int));

int pageFaults = 0;

int i, j, k, lru;

printf("\nLRU Page Replacement Policy:\n");

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

int flag = 0;

// Check if the page is already in the frame

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

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

flag = 1; // Page found in the frame

counter[j] = i; // Update the counter for the page

break;

}

}

// If page is not found in the frame

if (flag == 0) {

lru = 0;

// Find the least recently used page

for (j = 1; j < capacity; j++) {

if (counter[j] < counter[lru])

lru = j;

}

frame[lru] = pages[i];

counter[lru] = i;

pageFaults++;

}

// Print the current state of the frame

printf("Page %d: ", pages[i]);

for (k = 0; k < capacity; k++) {

if (frame[k] == -1)

printf("- ");

else

printf("%d ", frame[k]);

}

printf("\n");

}

printf("\nTotal Page Faults: %d\n", pageFaults);

free(frame);

free(counter);

}

int main() {

int n, capacity, i;

printf("Enter the number of pages: ");

scanf("%d", &n);

int \*pages = (int \*)malloc(n \* sizeof(int));

printf("Enter the page reference sequence: ");

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

scanf("%d", &pages[i]);

}

printf("Enter the number of frames: ");

scanf("%d", &capacity);

fifo(pages, n, capacity);

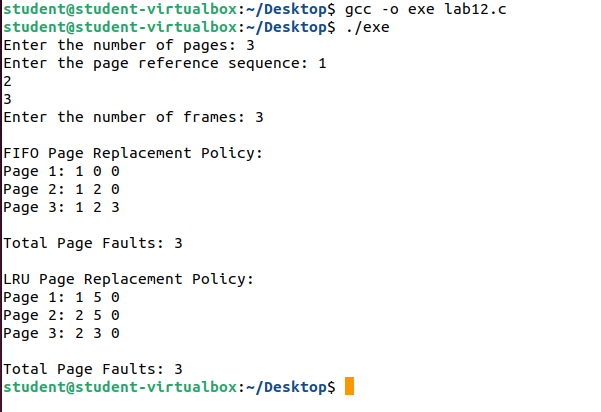
lru(pages, n, capacity);

free(pages);

return 0;

}

**Output:**



**Task No. 2:** Execute both programs for the same set of reference strings. What difference did you observe? Comment

**Solution:**

The FIFO (First-In-First-Out) page replacement policy replaces the oldest page in the frame when a page fault occurs. It maintains a queue or circular buffer of frames and evicts the page that was brought in first. FIFO is a simple and straightforward algorithm but does not consider the usage patterns of pages.

On the other hand, the LRU (Least Recently Used) page replacement policy replaces the page that has not been used for the longest time. It keeps track of the time when each page was last accessed and evicts the page that was accessed least recently. LRU attempts to make intelligent decisions by considering the historical usage patterns of pages.

When executing both programs for the same set of reference strings, you may observe that the total number of page faults differs between FIFO and LRU. The LRU policy generally performs better than FIFO in terms of reducing the number of page faults. This is because LRU takes into account the temporal locality principle, which states that recently accessed pages are more likely to be accessed again in the near future. By evicting the least recently used page, LRU aims to minimize the number of future page faults.

However, it's important to note that the performance of different page replacement policies can vary depending on the specific workload and access patterns. While LRU tends to provide better results in many cases, there may be scenarios where FIFO performs equally well or even outperforms LRU. It ultimately depends on the characteristics of the workload and the memory access patterns.