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

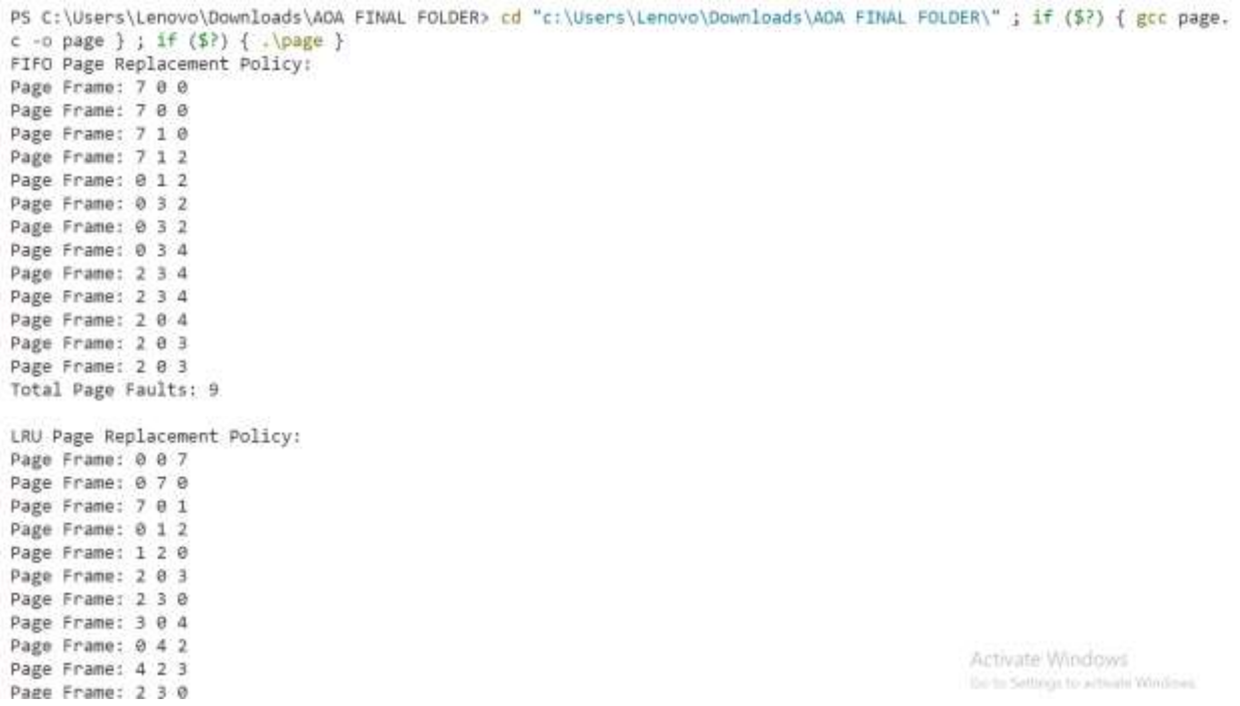
The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference.

Code:

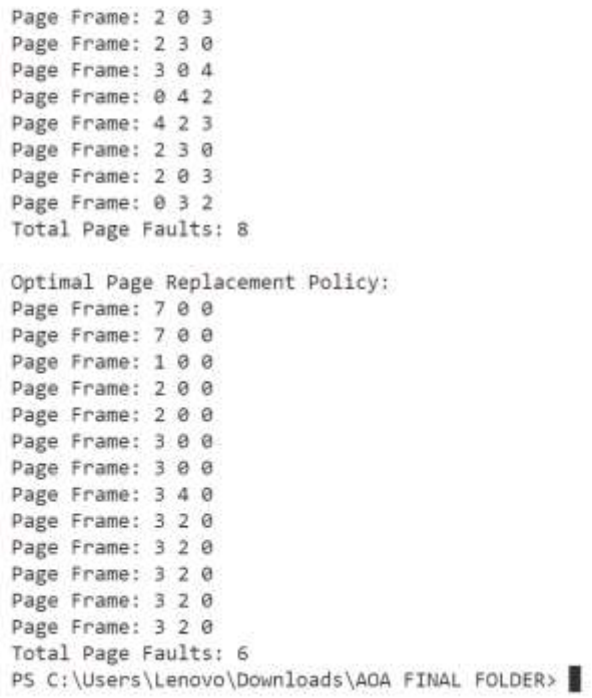
```
#include <stdio.h>
#include <stdlib.h>

#define FRAME_SIZE 3 // Size of the page frame

// Function prototypes
void fifo(int pages[], int n);
void lru(int pages[], int n);
void optimal(int pages[], int n);

int main() {
   int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2}; // Reference string
   int n = sizeof(pages) / sizeof(pages[0]);

   printf("FIFO Page Replacement Policy:\n");
   fifo(pages, n);

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

   printf("\nOptimal Page Replacement Policy:\n");
   optimal(pages, n);

   return 0;
}

void fifo(int pages[], int n) {
   int pageFrame[FRAME_SIZE] = {-1}; // Initialize page frame with -1 (empty)
   int pageFaults = 0;
   int index = 0; // Index for page frame replacement

   for (int i = 0; i < n; i++) {
        int page = pages[i];
        int found = 0;

        // Check if page is already in page frame
        for (int j = 0; j < FRAME_SIZE; j++) {
            if (pageFrame[j] == page) {
                found = 1;
                break;
            }
        }

        // If page not found in page frame, replace a page and increment page faults
        if (!found) {
            pageFrame[index] = page;
            index = (index + 1) % FRAME_SIZE;
            pageFaults++;
        }

        // Display current page frame
        printf("Page Frame: ");
        for (int j = 0; j < FRAME_SIZE; j++) {
            printf("%d ", pageFrame[j]);
        }
        printf("\n");
    }

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

void lru(int pages[], int n) {
    int pageFrame[FRAME_SIZE] = {-1}; // Initialize page frame with -1 (empty)
    int pageFaults = 0;
```

```
for (int i = 0; i < n; i++) {
    int page = pages[i];
    int found = 0;

    // Check if page is already in page frame
    for (int j = 0; j < FRAME_SIZE; j++) {
        if (pageFrame[j] == page) {
            found = 1;
            // Move the page to the end (most recently used)
            for (int k = j; k < FRAME_SIZE - 1; k++) {
                pageFrame[k] = pageFrame[k + 1];
            }
            pageFrame[FRAME_SIZE - 1] = page;
            break;
        }
    }

    // If page not found in page frame, replace a page and increment page faults
    if (!found) {
        // Shift all pages to the left to make room for the new page
        for (int j = 0; j < FRAME_SIZE - 1; j++) {
            pageFrame[j] = pageFrame[j + 1];
        }
        pageFrame[FRAME_SIZE - 1] = page;
        pageFaults++;
    }

    // Display current page frame
    printf("Page Frame: ");
    for (int j = 0; j < FRAME_SIZE; j++) {
        printf("%d ", pageFrame[j]);
    }
    printf("\n");
    }

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

void optimal(int pages[], int n) {
    int pageFrame[FRAME_SIZE] = {-1}; // Initialize page frame with -1 (empty)
    int pageFaults = 0;

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        int found = 0;

        // Check if page is already in page frame
        for (int j = 0; j < FRAME_SIZE; j++) {
            if (pageFrame[j] == page) {
                found = 1;
                break;
            }
        }

        // If page not found in page frame, find the page to replace with the longest
future distance
        if (!found) {
            int maxDistance = -1;
            int replaceIndex = -1;
            for (int j = 0; j < FRAME_SIZE; j++) {
                int distance = 0;
                for (int k = i + 1; k < n; k++) {
                    if (pageFrame[j] == pages[k]) {
                        break;
                    }
                    distance++;
            }
            if (distance > maxDistance) {
                maxDistance = distance;
                replaceIndex = j;
            }
        }
        pageFrame[replaceIndex] = page;
        pageFaults++;
    }

    // Display current page frame
    printf("Page Frame: ");
    for (int j = 0; j < FRAME_SIZE; j++) {
        printf("%d ", pageFrame[j]);
    }
    printf("\n");
  }

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

Output:



```
PS C:\Users\Lenovo\Downloads\AOA FINAL FOLDER> cd "c:\Users\Lenovo\Downloads\AOA FINAL FOLDER\" ; if ($?) { gcc page.
c -o page } ; if ($?) { .\page }
FIFO Page Replacement Policy:
Page Frame: 7 0 0
Page Frame: 7 0 0
Page Frame: 7 1 0
Page Frame: 7 1 2
Page Frame: 0 1 2
Page Frame: 0 3 2
Page Frame: 0 3 2
Page Frame: 0 3 4
Page Frame: 2 3 4
Page Frame: 2 3 4
Page Frame: 2 0 4
Page Frame: 2 0 3
Page Frame: 2 0 3
Total Page Faults: 9

LRU Page Replacement Policy:
Page Frame: 0 0 7
Page Frame: 0 7 0
Page Frame: 7 0 1
Page Frame: 0 1 2
Page Frame: 1 2 0
Page Frame: 2 0 3
Page Frame: 2 3 0
Page Frame: 3 0 4
Page Frame: 0 4 2
Page Frame: 4 2 3
Page Frame: 2 3 0
```



```
Page Frame: 2 0 3
Page Frame: 2 3 0
Page Frame: 3 0 4
Page Frame: 0 4 2
Page Frame: 4 2 3
Page Frame: 2 3 0
Page Frame: 2 0 3
Page Frame: 0 3 2
Total Page Faults: 8

Optimal Page Replacement Policy:
Page Frame: 7 0 0
Page Frame: 7 0 0
Page Frame: 1 0 0
Page Frame: 2 0 0
Page Frame: 2 0 0
Page Frame: 3 0 0
Page Frame: 3 0 0
Page Frame: 3 4 0
Page Frame: 3 2 0
Page Frame: 3 2 0
Page Frame: 3 2 0
Page Frame: 3 2 0
Page Frame: 3 2 0
Total Page Faults: 6
PS C:\Users\Lenovo\Downloads\AOA FINAL FOLDER>
```

Conclusion:
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.

Why do we need page replacement strategies?

Page replacement strategies are essential in computer memory management for several reasons:

1. Limited Physical Memory: In modern computer systems, physical memory (RAM) is limited. When the demand for memory exceeds the available physical memory, the operating system needs to use secondary storage (like hard disk) to store less frequently accessed data. Page replacement strategies determine which pages should be swapped out of physical memory to make room for new pages.
2. Handling Page Faults: Page faults occur when a program tries to access a page that is not currently present in physical memory. Page replacement strategies dictate how the operating system selects which page to evict from memory when a page fault occurs.
3. Optimizing Performance: Effective page replacement strategies aim to minimize the number of page faults and optimize overall system performance. By keeping frequently accessed pages in memory and evicting less frequently accessed or unused pages, page replacement strategies help

reduce the overhead associated with accessing data from slower secondary storage devices.

4. Managing Memory Utilization: Page replacement strategies help manage memory utilization efficiently. They ensure that memory resources are allocated to processes that need them the most, thereby maximizing the utilization of available memory and preventing unnecessary waste.
5. Adapting to Changing Workloads: Workloads and memory access patterns can vary over time. Page replacement strategies should be flexible and adaptable to different workloads, ensuring that the system can dynamically adjust its memory management approach based on current demands.
6. Preventing Thrashing: Thrashing occurs when the system spends a significant amount of time swapping pages in and out of memory, resulting in poor performance. Effective page replacement strategies help prevent thrashing by making intelligent decisions about which pages to keep in memory and which ones to swap out.