VIRTUAL MEMORY OPTIMIZATION TOOL

OM GUPTA(1RV22CS133), RAHEEL JAWED(1RV22CS155),
N SASIDAR(1RV22CS123)

UNDER THE GUIDENCE OF :-

DR. JYOTI SHETTY (ASSISTANT PROFESSOR, DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

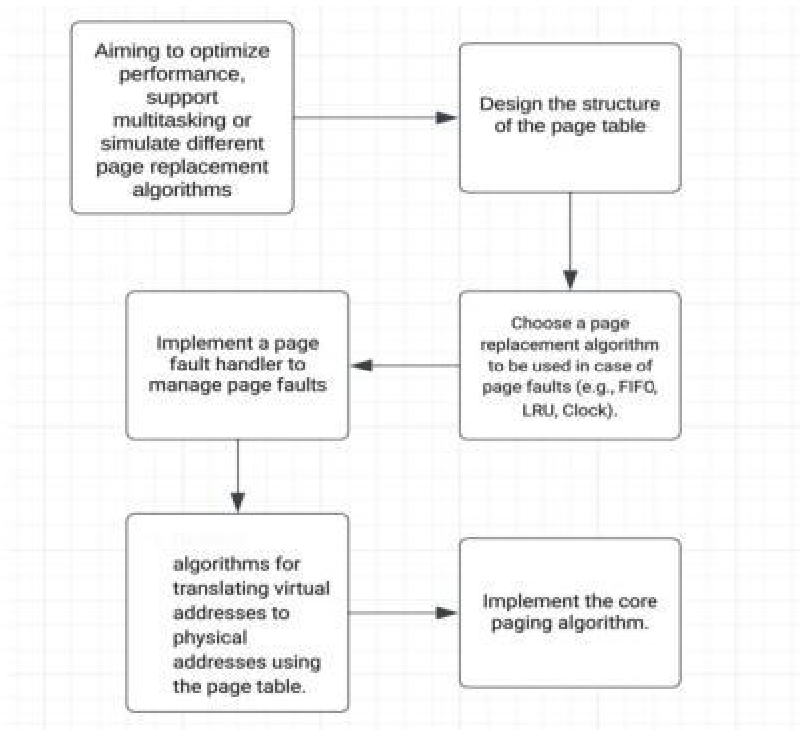
Introduction

- Virtual memory is a storage allocation system wherein secondary memory is accessed as if it were an extension of the primary memory, creating the perception of a substantially larger main memory for the user.
- Paging is a virtual memory method that divides memory into segments known as paging files. When a computer exhausts its RAM capacity, it moves any inactive pages to a designated area on the hard drive used for virtual memory. Rather than loading a single large process into primary memory, the operating system loads various segments of multiple processes. This boosts the level of multiprogramming, consequently enhancing CPU utilization.

Objective

Effective memory management is essential for operating systems to achieve optimal performance. Page replacement algorithms are central in deciding which pages remain in primary memory and which are moved to secondary storage. This project seeks to conduct a thorough examination and comparison of diverse page replacement algorithms, assessing their performance across different scenarios.

Methodology



Literature survey

PAPER 1: Experimental Quantum Secure Direct Communication with Single Photons

AUTHORS J. Yin, T.Y. Chen, Z.W. Yu, H. Li, X. Ma, Y. Liu, L. M. Duan, J.W. Pan

SUMMARY:-Quantum cure direct communication is an important mode of quantum communication in which secret messages are securely communicated directly over a quantum channel. Quantum secure direct communication is also a basic cryptographic primitive for constructing other quantum communication tasks, such as quantum authentication and quantum dialog.

Concepts and API Calls Used

1.**Memory Management**:

•Page Replacement: Both FIFO and Clock algorithms simulate different strategies for page replacement.

•Page Frame: A page frame represents a fixed-size portion of physical memory where pages are stored. The program simulates the allocation and replacement of pages within these frames.

2. **Process Management**:

•Execution Time Measurement: The program measures the execution time of each algorithm using the clock() function.

•Concurrency and Parallelism: Multiple processes maybe running concurrently, each requiring memory resources and potentially invoking page replacement algorithms.

3.**System Calls**:

•Memory Allocation and Deallocation: While not used directly in this program, memory allocation and deallocation system calls such as malloc() and free() are essential for managing memory resources in real-world applications.

•File I/O Operations: Although not directly relevant to page replacement, file I/O system calls are fundamental for interacting with the file system, which can indirectly impact memory usage and page replacement strategies.

4.**Algorithm Design**: The program demonstrates the implementation and evaluation of different scheduling algorithms (FIFO and Clock) for managing page replacement.

Code

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

#define FRAME_SIZE 3

typedef struct
{
    int page;
    int referenced;
} PageEntry;

int fifoPageFaults(int pages[], int n)
{
    PageEntry frame[FRAME_SIZE];
    int i, j, k, pos = 0;
    int page_faults = 0;

    for (i = 0; i < FRAME_SIZE; i++)
    {
        frame[i].page = -1;
        frame[i].referenced = 0;
    }

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < FRAME_SIZE; j++)
        {
            if (frame[j].page == pages[i])
                break;
        }

        if (j == FRAME_SIZE)
        {
            frame[pos].page = pages[i];
            page_faults++;
```

```c
            printf("FIFO Frame: ");
            for (k = 0; k < FRAME_SIZE; k++)
            {
                if (frame[k].page == -1)
                    printf(" - ");
                else
                    printf("%2d ", frame[k].page);
            }
            printf("\n");
        }

    return page_faults;
}

int clockPageFaults(int pages[], int n)
{
    PageEntry frame[FRAME_SIZE];
    int i, j, k, pos = 0;
    int page_faults = 0;

    for (i = 0; i < FRAME_SIZE; i++)
    {
        frame[i].page = -1;
        frame[i].referenced = 0;
    }

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < FRAME_SIZE; j++)
        {
            if (frame[j].page == pages[i])
            {
                frame[j].referenced = 1;
                break;
            }
        }
```

```c
        if (j == FRAME_SIZE)
        {
            while (frame[pos].referenced == 1)
            {
                frame[pos].referenced = 0;
                pos = (pos + 1) % FRAME_SIZE;
            }

            frame[pos].page = pages[i];
            frame[pos].referenced = 0;
            page_faults++;

            pos = (pos + 1) % FRAME_SIZE;
        }

        printf("Clock Frame: ");
        for (k = 0; k < FRAME_SIZE; k++)
        {
            if (frame[k].page == -1)
                printf(" - ");
            else
                printf("%2d ", frame[k].page);
        }
        printf("\n");
    }

    return page_faults;
}

int main()
{
    int pages[] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}; // Reference string
    int n = sizeof(pages) / sizeof(pages[0]);

    printf("Reference String: ");
    for (int i = 0; i < n; i++)
        printf("%d ", pages[i]);
```

```c
    printf("Reference String: ");
    for (int i = 0; i < n; i++)
        printf("%d ", pages[i]);
    printf("\n\n");

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    int fifo_faults = fifoPageFaults(pages, n);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nFIFO Page Faults: %d\n", fifo_faults);
    printf("FIFO Execution Time: %f seconds\n\n", cpu_time_used);

    start = clock();
    int clock_faults = clockPageFaults(pages, n);
    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nClock Page Faults: %d\n", clock_faults);
    printf("Clock Execution Time: %f seconds\n\n", cpu_time_used);

    return 0;
}
```

.

Output:

```
Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

FIFO Frame:  1  -  -
FIFO Frame:  1  2  -
FIFO Frame:  1  2  3
FIFO Frame:  4  2  3
FIFO Frame:  4  1  3
FIFO Frame:  4  1  2
FIFO Frame:  5  1  2
FIFO Frame:  5  1  2
FIFO Frame:  5  1  2
FIFO Frame:  5  3  2
FIFO Frame:  5  3  4
FIFO Frame:  5  3  4

FIFO Page Faults: 9
FIFO Execution Time: 0.003000 seconds

Clock Frame:  1  -  -
Clock Frame:  1  2  -
Clock Frame:  1  2  3
Clock Frame:  4  2  3
Clock Frame:  4  1  3
Clock Frame:  4  1  2
Clock Frame:  5  1  2
Clock Frame:  5  1  2
Clock Frame:  5  1  2
```

Conclusion:

FIFO algorithm is more efficient than CLOCK algorithm as there are less page faults. The execution time of CLOCK algorithm is faster.