Virtual Memory Management

21CSC202J - OPERATING SYSTEMS

A MINI-PROJECT REPORT

Submitted By

Yashwanth R [RA2311003020173]

Balajee E [RA2311003020143]

Harihara Alagappan V [RA2311003020168]

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

RAMAPURAM CAMPUS, CHENNAI-600089

OCTOBER 2024

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
(Deemed to be University U/S 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this mini project report titled **“VIRTUAL MEMORY MANAGEMENT SYSTEM”** is the bonafide work of **“YASHWANTH R (RA2311003020173) , BALAJEE E (RA2311003020143) , HARIHARA ALAGAPPAN V (RA2311003020168)”**

SIGNATURE

Dr.M.S.ANTONY VIGIL

Assistant Professor (Sr.G),

Department of Computer Science and Engineering,
SRM Institute of Science and Technology,

Ramapuram Campus, Chennai.

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

RAMAPURAM, CHENNAI -600089

DECLARATION

We hereby declare that the entire work contained in this mini project report titled "**VIRTUAL MEMORY MANAGEMENT SYSTEM"** has been carried out by **YASHWANTH R [RA2311003020173], BALAJEE E [RA2311003020143] , HARIHARA ALAGAPPAN V [RA2311003020168]** at SRM Institute of Science and Technology, Ramapuram Campus, Chennai - 600089,

Place: Chennai

Date:

ABSTRACT

This project presents a comprehensive simulation of virtual memory management within an operating system, focusing on the implementation of page tables and page replacement policies. Virtual memory is a fundamental concept that allows systems to use disk space as an extension of RAM, enabling applications to operate efficiently beyond physical memory constraints. The project includes the development of a page table, which maps virtual addresses to physical frames, maintaining the status of each page (valid or invalid) to facilitate seamless access. Additionally, we implement a page replacement policy, specifically the FIFO (First-In-First-Out) algorithm, to manage memory efficiently when physical frames are exhausted. Through a user-friendly interface, users can interactively create page tables, trigger demand paging, and observe the system's behaviour during page faults and replacements. This project aims to enhance understanding of virtual memory mechanisms, providing insights into the complexities of memory management in modern operating systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

FIFO	First In First Out
OPR	Optimal Page Replacement
LRU	Least Recently Used
MRU	Most Recently Used

CHAPTER 1

INTRODUCTION

1.1 Overview

Virtual memory is a powerful feature of modern operating systems that allows them to use disk space as an extension of RAM (Random Access Memory). This capability addresses the limitations of physical memory, enabling systems to run larger applications and manage multiple processes concurrently, even when the total memory requirements exceed the available physical RAM.

One of the key functions of virtual memory is providing each process with its own isolated address space. This means that each program operates in a separate memory area, making it appear as though it has full access to a large and contiguous block of memory. This isolation not only improves security but also enhances stability, as one process cannot directly access the memory of another process.

In addition to isolation, virtual memory facilitates efficient memory utilization. By using disk space as a supplement to physical memory, the system can maintain frequently accessed pages in RAM while moving less frequently used data to disk. This optimization ensures that the overall memory footprint is managed effectively, allowing for more applications to run simultaneously.

The management of memory is further simplified by virtual memory, as programmers do not need to handle memory allocation and deallocation manually. The operating system takes on this responsibility, reducing the likelihood of memory-related bugs, such as memory leaks or segmentation faults, which can arise in environments where manual memory management is required.

Another critical aspect of virtual memory is its ability to handle situations where physical memory is fully utilized. When this occurs, the virtual memory manager can swap out less frequently accessed pages to disk in a process known as paging. This allows the system to continue functioning smoothly, providing the flexibility to run multiple applications without crashing due to memory limitations.

Finally, virtual memory plays a vital role in supporting multiprogramming. It enables multiple processes to run concurrently by ensuring that they do not interfere with each other's memory. This is crucial for the performance of multi-user systems and for applications that demand high responsiveness.

Overall, the virtual memory manager is a fundamental aspect of how operating systems manage memory, enhancing both performance and usability. By abstracting the complexity of physical memory management, it provides a seamless experience for users and developers alike.

1.2 Problem Statement

The increasing complexity and size of modern applications necessitate efficient memory management systems. Traditional physical memory is often insufficient to handle large processes, leading to performance bottlenecks and inefficient resource utilization. There is a need for a virtual memory management system that can abstract physical memory, manage address translations, handle page faults, and efficiently allocate and deallocate memory.

1.3 Aim of the Project

The aim of this project is to design and implement a comprehensive virtual memory management system within an operating system. This system will provide an efficient and transparent way to extend the available physical memory, supporting larger and more complex processes while ensuring optimal performance and resource utilization.

1.4 Scope of the Project

The scope of this project involves designing a virtual memory system that abstracts physical memory from processes, allowing them to operate as if they have access to a contiguous block of memory while the operating system manages the actual allocation. It includes developing a data structure to efficiently manage the mapping between virtual and physical memory, ensuring seamless address translation through both hardware and software mechanisms. Additionally, the project will implement mechanisms to handle page faults, loading necessary pages from secondary storage into memory when accessed by processes. Finally, it will incorporate algorithms for efficient memory allocation and deallocation, optimizing the use of physical memory by allocating it to processes as needed and reclaiming it when no longer in use.

CHAPTER 2

SOFTWARE AND HARDWARE SPECIFICATIONS

2.1 Hardware Requirements

The equipment prerequisites for this venture are as takes after:

•Processor: Intel Core i3 or equivalent (Intel Core i5 or equivalent for better performance)

• Ethernet association (LAN) OR a remote connector (Wi-Fi)

• Difficult Drive: Least 100 GB; Suggested 200 GB or more

• Memory (RAM): 4 GB RAM (8 GB RAM or more, especially for multitasking)

2. Software Requirements

The program prerequisites for this venture are as takes after:

- **Operating System**:
  - Compatible with Windows, macOS, or Linux distributions (Ubuntu, Fedora, etc.)
- **Development Environment**:
  - C/C++ Compiler (e.g., GCC for Linux, MinGW for Windows)
  - Integrated Development Environment (IDE) (e.g., Code::Blocks, Visual Studio, or any text editor)
- **Libraries**:
  - Standard C libraries (usually included with the compiler)
- **Additional Tools** (Optional):

- Version control software (e.g., Git) for managing project versions
- Debugging tools to assist in troubleshooting the program

CHAPTER 3

PROJECT DESCRIPTION

3.1 Introduction

Virtual Memory

Virtual memory is a memory management technique used by operating systems to create the illusion of a large, continuous block of memory for applications, even if the physical memory (RAM) is limited. This technique allows the system to compensate for physical memory shortages, enabling larger applications to run on systems with less RAM.

Paging

Paging is a memory management technique used in operating systems to manage how memory is allocated to processes. In this method, both the physical memory (RAM) and the logical memory (used by processes) are divided into fixed-size blocks. The blocks in physical memory are called frames, and the blocks in logical memory are called pages.

When a process needs to access memory, the operating system allocates the required number of frames to the process and maps its pages to these frames. This mapping is maintained in a data structure called the page table, which helps translate logical addresses to physical addresses.

Paging allows processes to use more memory than is physically available by swapping pages in and out of secondary storage (like a hard drive) as needed. This technique helps improve performance, optimize resource utilization, and reduce the risk of page faults, which occur when a process tries to access a page not currently in physical memory.

Page Tables

Page tables are a crucial data structure used in virtual memory systems within operating systems to manage the mapping between virtual addresses and physical addresses.

Role of Page Tables

1. Mapping Virtual to Physical Addresses:

   Page tables store the mappings between virtual addresses (used by processes) and physical addresses (used by the hardware, specifically RAM). This allows processes to use virtual addresses without needing to know the actual physical location of the data.

2. Address Translation:

   When a process accesses memory, the CPU generates a virtual address. The page table is used to translate this virtual address into a physical address. This translation is essential for the CPU to access the correct location in physical memory.

3. Handling Non-Contiguous Memory:

   Page tables enable the operating system to allocate physical memory in a non-contiguous manner. This means that even if a process thinks it has a large, contiguous block of memory, the actual physical memory can be scattered across different locations.

4. Page Table Entries (PTEs):

   Each entry in a page table, known as a Page Table Entry (PTE), contains information about a specific page, such as the physical address of the page frame, access permissions, and status bits (e.g., whether the page is in memory or on disk).

Page Fault

A page fault happens when a running program accesses a memory page that is mapped into the virtual address space but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of a page fault, the Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Page Replacement Algorithms

Page replacement algorithms are techniques used in operating systems to manage memory efficiently when the virtual memory is full. When a new page needs to be loaded into physical memory and there is no free space, these algorithms determine which existing page to replace.

Common Page Replacement Techniques

- First In First Out (FIFO)
- Optimal Page replacement
- Least Recently Used
- Most Recently Used (MRU)

1) First In First Out

The First-In-First-Out (FIFO) page replacement algorithm is a straightforward method used in operating systems to manage memory. It operates by maintaining a queue of pages currently in memory, where the oldest page is at the front and the newest at the back. When a new page needs to be loaded and memory is full, the page at the front of the queue (the oldest) is replaced. This simplicity makes FIFO easy to implement and understand, but it often leads to suboptimal performance, as it does not consider the frequency or recency of page usage. This can result in frequent page faults, especially if the oldest page is still in active use, a phenomenon known as Belady’s anomaly. Despite its predictability and minimal bookkeeping requirements, FIFO’s lack of adaptability to actual usage patterns makes it less efficient compared to more sophisticated algorithms like Least Recently Used (LRU) or Optimal Page Replacement.

Example 1: Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames. Find the number of page faults.

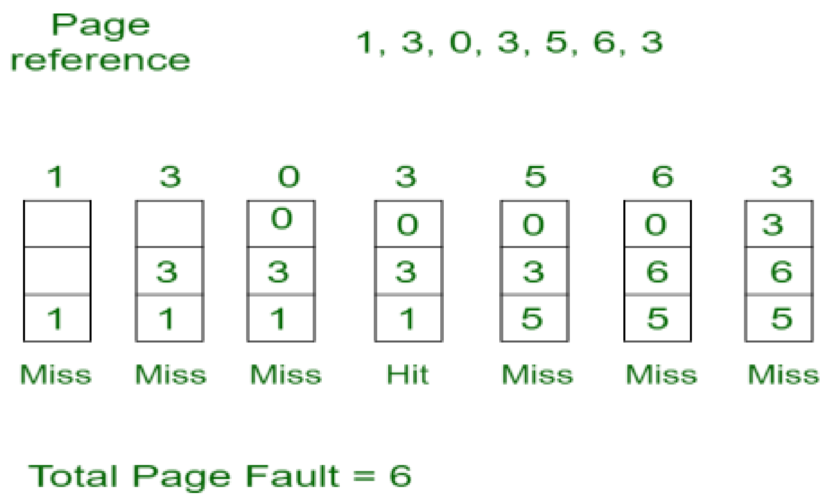


Fig 3.1 FIFO Algorithm

2) Optimal Page replacement

The Optimal Page Replacement algorithm is a theoretical approach used in operating systems to manage memory by minimizing the number of page faults. It works by replacing the page that will not be used for the longest period in the future. This algorithm requires knowledge of future memory references, which makes it impractical for real-world implementation but serves as a benchmark for evaluating other algorithms. By always selecting the page that will be needed the furthest in the future, the Optimal Page Replacement algorithm ensures the lowest possible page fault rate. However, its reliance on future knowledge limits its practical application, as predicting future memory access patterns accurately is not feasible. Despite this, it provides a valuable standard for comparing the efficiency of other page replacement strategies.

Example-2: Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frame. Find number of page fault.

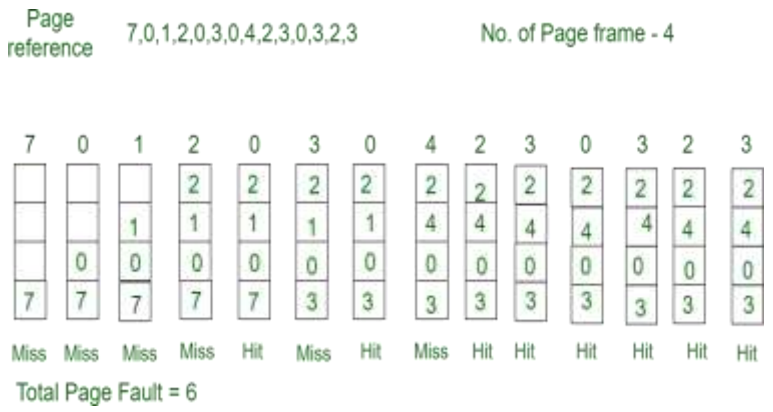
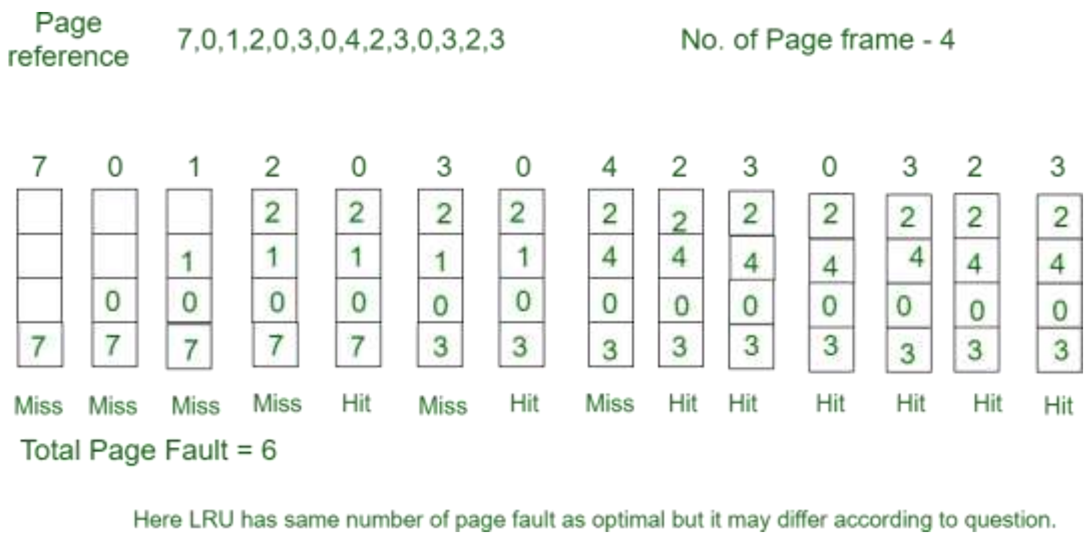


Fig 3.2 OPR Algorithm

3) **Least Recently Used**

The Least Recently Used (LRU) page replacement algorithm is a widely used method in operating systems for managing memory efficiently. LRU operates on the principle that pages that have not been used for the longest time are the least likely to be needed soon. To implement LRU, the system maintains a record of the order in which pages are accessed. When a new page needs to be loaded and memory is full, the page that has not been used for the longest period is replaced. This approach approximates the optimal page replacement strategy by leveraging past behaviour to predict future needs.

Example-3: Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames. Find number of page faults



Page reference
7,0,1,2,0,3,0,4,2,3,0,3,2,3
No. of Page frame - 4
Miss Miss Miss Miss Hit Miss Hit Miss Hit Hit Hit Hit Hit Hit
Total Page Fault = 6
Here LRU has same number of page fault as optimal but it may differ according to question.

Fig 3.3 LRU Algorithm

4) **Most Recently Used**

The Most Recently Used (MRU) page replacement algorithm is a strategy used in operating systems to manage memory by replacing the page that has been accessed most recently. This approach is based on the assumption that the most recently used page is likely to be accessed again soon, which can be beneficial in certain access patterns. However, this assumption often contradicts typical usage patterns where recently accessed pages are more likely to be needed again soon, making MRU less effective in many scenarios.

Example-4: Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frame. Find number of page fault.

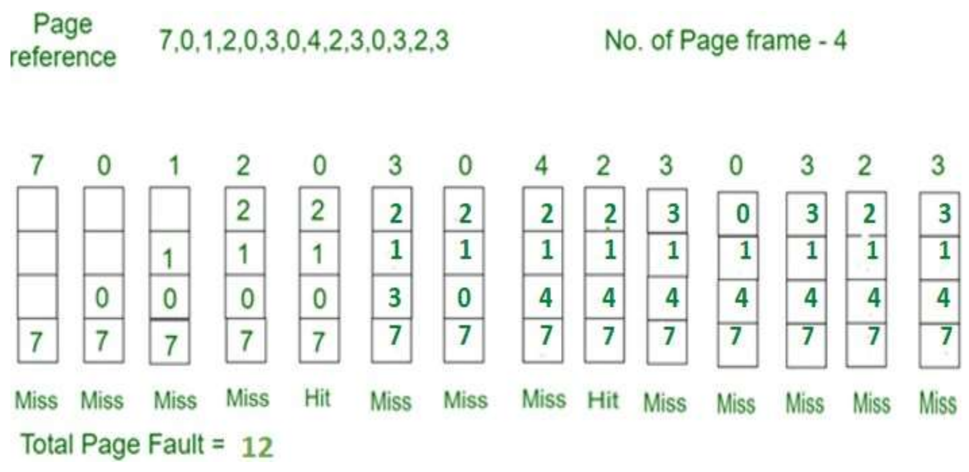


Fig 3.4 MRU Algorithm

2. Algorithm Used

The program simulates a virtual memory management system, incorporating fundamental concepts such as page tables, demand paging, and page replacement strategies. Virtual memory is crucial in modern operating systems as it allows processes to use more memory than is physically available, enhances multitasking, and isolates process memory spaces. The program provides a user-friendly interface for demonstrating these memory management techniques, showcasing how virtual addresses are mapped to physical memory frames, how pages are loaded on demand, and how memory is managed when physical resources are exhausted.

Page Table Management:

- **Initialization**: The page table is initialized to mark all entries as invalid. This ensures that any virtual address access before mapping will trigger a page fault.
- **Mapping**: When a page is loaded, its entry in the page table is updated to reflect its valid status and the corresponding physical frame.

Demand Paging:

- **Page Fault Detection**: When accessing a virtual address, the program checks if the corresponding page is in physical memory. If not, it triggers a page fault.
- **Page Loading**: The program loads the required page into physical memory, either into a free frame or by evicting an existing page based on FIFO policy.

FIFO Page Replacement:

- **Page Replacement Strategy**: If physical memory is full, the program evicts the oldest page (the first one added) and replaces it with the new page. This is done using a queue to track the order of pages in memory.

User Interface:

- The program provides a menu-driven interface, allowing users to create a page table, perform demand paging, read from virtual addresses, or execute page replacements.

3. Advantages of Algorithm used

Comprehensive Simulation:

- Integrates multiple memory management concepts into one program, providing a complete view of virtual memory systems.

Educational Value:

- Serves as an excellent teaching tool for understanding how virtual memory works, making it easier for students to grasp complex concepts.

User-Friendly Interface:

- The menu-driven approach allows for easy interaction and experimentation with different functionalities without requiring extensive command-line knowledge.

Extensibility:

- The modular design makes it easy to extend the program by adding more page replacement algorithms, enhancing demand paging features, or improving data handling methods.

Realistic Simulation:

- Simulates real-world memory management scenarios, demonstrating how operating systems handle memory allocation and paging.

4. Architecture

1. **User Interface**:
    - This is where the user interacts with the program. It provides a menu with options for various functionalities like creating a page table, performing demand paging, reading from a virtual address, and executing page replacement.
2. **Memory Management Module**:
    - **Page Table**:
        - Contains entries that map virtual pages to physical frames. Each entry has a validity flag and a frame number.

- **Physical Memory**:
    - Simulated physical memory organized into frames. Each frame holds a block of data corresponding to a page.
- **Page Queue**:
    - A FIFO queue that keeps track of the order of pages in physical memory for managing page replacement.

3. **Page Fault Handling**:
    - This component is activated when a page fault occurs (i.e., when a virtual address accesses a page that is not in physical memory). It:
        - Checks for free frames.
        - Loads the required page into a frame.
        - Implements page replacement using FIFO if no free frames are available.
4. **Data Flow**:
    - **User Input**: The user selects options from the menu, triggering functions in the memory management module.
    - **Address Translation**: When accessing a virtual address, the system checks the page table to determine if the page is valid. If not, a page fault is triggered, and the handling process starts.
    - **Page Loading**: If a page is not present in memory, it is loaded, and the page table is updated accordingly.

3.5 Explanation of Project

Purpose:

- The program is designed to manage memory in a way that allows applications to use more memory than is physically available on the system. It does this by using techniques like paging, where memory is divided into fixed-size blocks.

Key Components:

- **Page Table**: This data structure maps virtual memory pages to physical memory frames. It keeps track of which pages are currently loaded in memory and their respective frame locations.
- **Physical Memory**: Simulated as an array of frames, each frame holds data corresponding to a virtual page.
- **Page Queue**: A FIFO (First-In-First-Out) queue that tracks the order in which pages are loaded into physical memory, which helps in managing page replacements when memory is full.

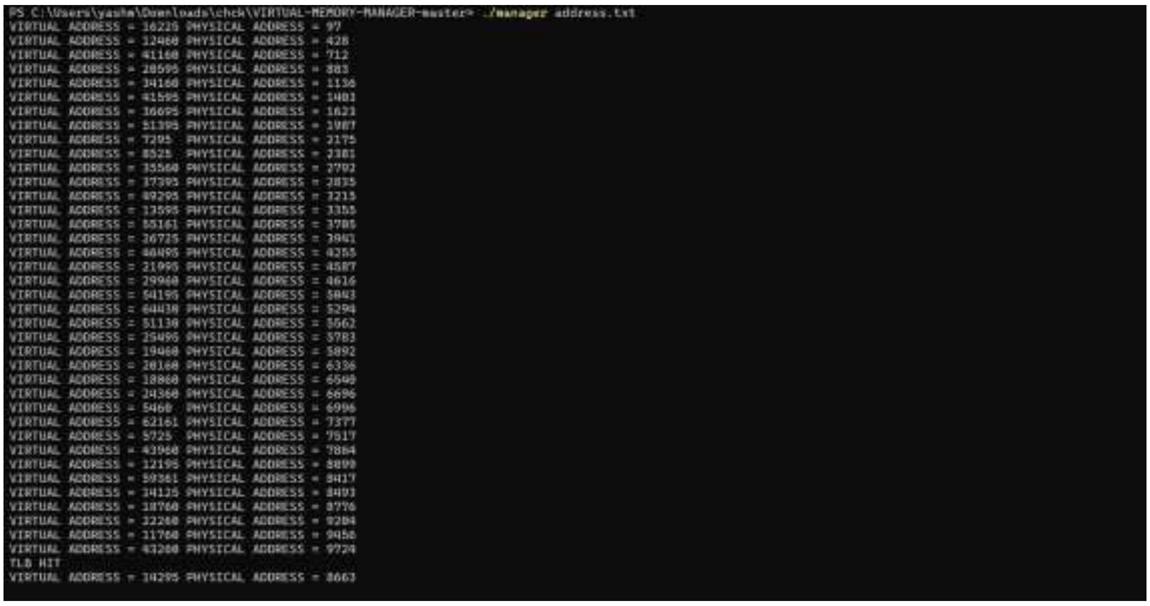
Core Functionality:

- The user interacts with the program through a menu that allows them to create a page table, perform demand paging (loading pages on first access), read from virtual addresses, and execute page replacements.
- When a program accesses a virtual address, the program checks if the corresponding page is in memory. If not, it triggers a page fault, leading to the loading of the required page into memory. If memory is full, the program replaces the oldest page using the FIFO strategy.

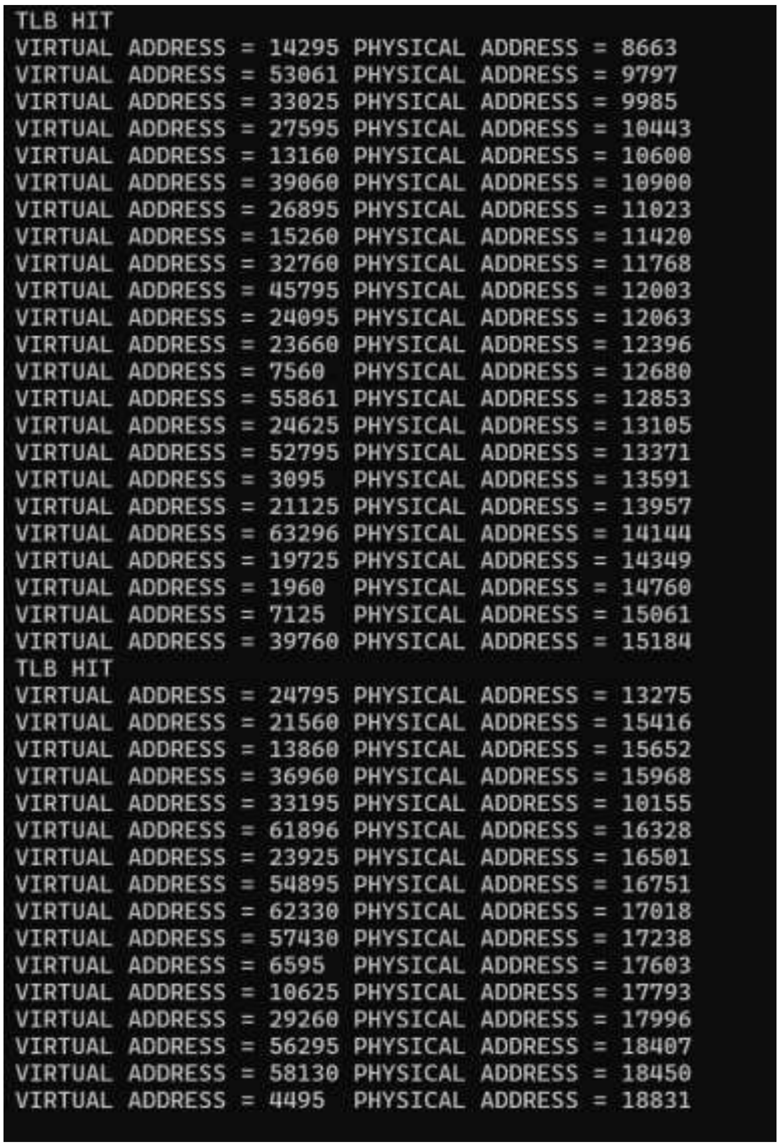
User Interaction:

- Users can enter virtual addresses to see how the system handles memory access, including page faults and the loading of pages. This provides a hands-on understanding of how virtual memory works in practice.

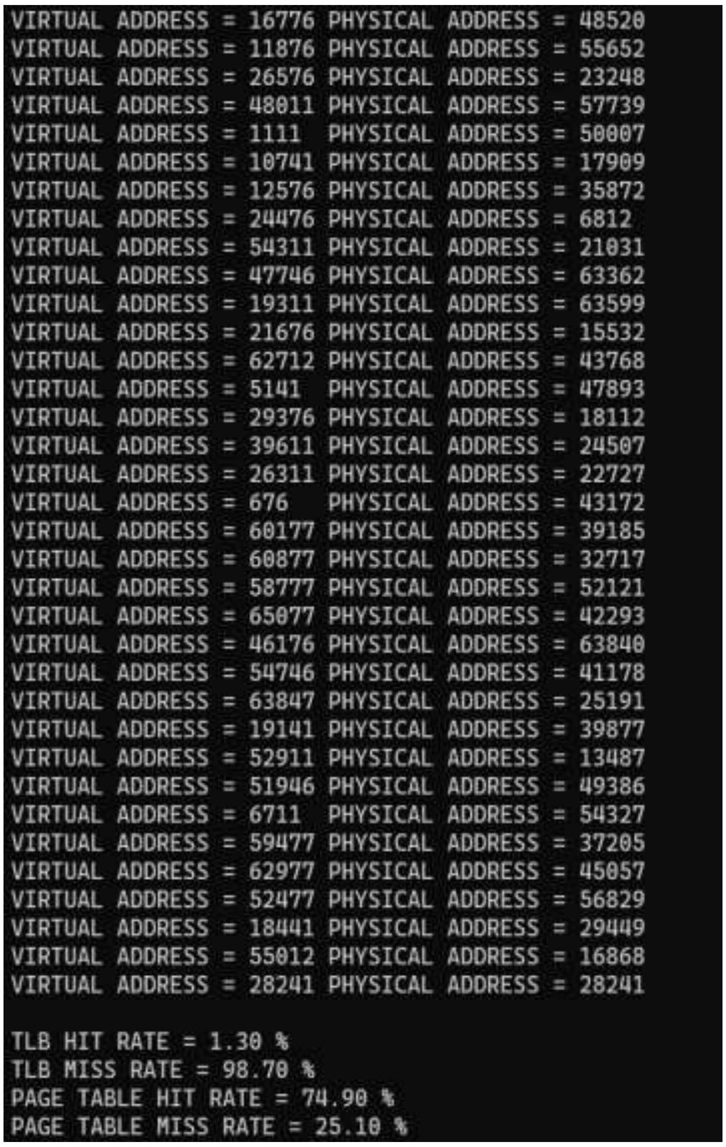
OUTPUT



```
PS C:\Users\yashm\Downloads\chck\VIRTUAL-MEMORY-MANAGER-master> ./manager address.txt
VIRTUAL ADDRESS = 16225 PHYSICAL ADDRESS = 97
VIRTUAL ADDRESS = 12460 PHYSICAL ADDRESS = 428
VIRTUAL ADDRESS = 41160 PHYSICAL ADDRESS = 712
VIRTUAL ADDRESS = 28595 PHYSICAL ADDRESS = 883
VIRTUAL ADDRESS = 34160 PHYSICAL ADDRESS = 1136
VIRTUAL ADDRESS = 41595 PHYSICAL ADDRESS = 1403
VIRTUAL ADDRESS = 36695 PHYSICAL ADDRESS = 1623
VIRTUAL ADDRESS = 51395 PHYSICAL ADDRESS = 1987
VIRTUAL ADDRESS = 7295  PHYSICAL ADDRESS = 2175
VIRTUAL ADDRESS = 8525  PHYSICAL ADDRESS = 2381
VIRTUAL ADDRESS = 35560 PHYSICAL ADDRESS = 2792
VIRTUAL ADDRESS = 37395 PHYSICAL ADDRESS = 2835
VIRTUAL ADDRESS = 49295 PHYSICAL ADDRESS = 3215
VIRTUAL ADDRESS = 13595 PHYSICAL ADDRESS = 3355
VIRTUAL ADDRESS = 55161 PHYSICAL ADDRESS = 3785
VIRTUAL ADDRESS = 26725 PHYSICAL ADDRESS = 3941
VIRTUAL ADDRESS = 46495 PHYSICAL ADDRESS = 4255
VIRTUAL ADDRESS = 21995 PHYSICAL ADDRESS = 4587
VIRTUAL ADDRESS = 29960 PHYSICAL ADDRESS = 4616
VIRTUAL ADDRESS = 54195 PHYSICAL ADDRESS = 5043
VIRTUAL ADDRESS = 64438 PHYSICAL ADDRESS = 5294
VIRTUAL ADDRESS = 51130 PHYSICAL ADDRESS = 5562
VIRTUAL ADDRESS = 25495 PHYSICAL ADDRESS = 5783
VIRTUAL ADDRESS = 19460 PHYSICAL ADDRESS = 5892
VIRTUAL ADDRESS = 28160 PHYSICAL ADDRESS = 6336
VIRTUAL ADDRESS = 18860 PHYSICAL ADDRESS = 6540
VIRTUAL ADDRESS = 24360 PHYSICAL ADDRESS = 6696
VIRTUAL ADDRESS = 5460  PHYSICAL ADDRESS = 6996
VIRTUAL ADDRESS = 62161 PHYSICAL ADDRESS = 7377
VIRTUAL ADDRESS = 5725  PHYSICAL ADDRESS = 7517
VIRTUAL ADDRESS = 43960 PHYSICAL ADDRESS = 7864
VIRTUAL ADDRESS = 12195 PHYSICAL ADDRESS = 8099
VIRTUAL ADDRESS = 59361 PHYSICAL ADDRESS = 8417
VIRTUAL ADDRESS = 34125 PHYSICAL ADDRESS = 8493
VIRTUAL ADDRESS = 18760 PHYSICAL ADDRESS = 8776
VIRTUAL ADDRESS = 22260 PHYSICAL ADDRESS = 9204
VIRTUAL ADDRESS = 11760 PHYSICAL ADDRESS = 9458
VIRTUAL ADDRESS = 43260 PHYSICAL ADDRESS = 9724
TLB HIT
VIRTUAL ADDRESS = 14295 PHYSICAL ADDRESS = 8663
```



```
TLB HIT
VIRTUAL ADDRESS = 14295 PHYSICAL ADDRESS = 8663
VIRTUAL ADDRESS = 53061 PHYSICAL ADDRESS = 9797
VIRTUAL ADDRESS = 33025 PHYSICAL ADDRESS = 9985
VIRTUAL ADDRESS = 27595 PHYSICAL ADDRESS = 10443
VIRTUAL ADDRESS = 13160 PHYSICAL ADDRESS = 10600
VIRTUAL ADDRESS = 39060 PHYSICAL ADDRESS = 10900
VIRTUAL ADDRESS = 26895 PHYSICAL ADDRESS = 11023
VIRTUAL ADDRESS = 15260 PHYSICAL ADDRESS = 11420
VIRTUAL ADDRESS = 32760 PHYSICAL ADDRESS = 11768
VIRTUAL ADDRESS = 45795 PHYSICAL ADDRESS = 12003
VIRTUAL ADDRESS = 24095 PHYSICAL ADDRESS = 12063
VIRTUAL ADDRESS = 23660 PHYSICAL ADDRESS = 12396
VIRTUAL ADDRESS = 7560  PHYSICAL ADDRESS = 12680
VIRTUAL ADDRESS = 55861 PHYSICAL ADDRESS = 12853
VIRTUAL ADDRESS = 24625 PHYSICAL ADDRESS = 13105
VIRTUAL ADDRESS = 52795 PHYSICAL ADDRESS = 13371
VIRTUAL ADDRESS = 3095  PHYSICAL ADDRESS = 13591
VIRTUAL ADDRESS = 21125 PHYSICAL ADDRESS = 13957
VIRTUAL ADDRESS = 63296 PHYSICAL ADDRESS = 14144
VIRTUAL ADDRESS = 19725 PHYSICAL ADDRESS = 14349
VIRTUAL ADDRESS = 1960  PHYSICAL ADDRESS = 14760
VIRTUAL ADDRESS = 7125  PHYSICAL ADDRESS = 15061
VIRTUAL ADDRESS = 39760 PHYSICAL ADDRESS = 15184
TLB HIT
VIRTUAL ADDRESS = 24795 PHYSICAL ADDRESS = 13275
VIRTUAL ADDRESS = 21560 PHYSICAL ADDRESS = 15416
VIRTUAL ADDRESS = 13860 PHYSICAL ADDRESS = 15652
VIRTUAL ADDRESS = 36960 PHYSICAL ADDRESS = 15968
VIRTUAL ADDRESS = 33195 PHYSICAL ADDRESS = 10155
VIRTUAL ADDRESS = 61896 PHYSICAL ADDRESS = 16328
VIRTUAL ADDRESS = 23925 PHYSICAL ADDRESS = 16501
VIRTUAL ADDRESS = 54895 PHYSICAL ADDRESS = 16751
VIRTUAL ADDRESS = 62330 PHYSICAL ADDRESS = 17018
VIRTUAL ADDRESS = 57430 PHYSICAL ADDRESS = 17238
VIRTUAL ADDRESS = 6595  PHYSICAL ADDRESS = 17603
VIRTUAL ADDRESS = 10625 PHYSICAL ADDRESS = 17793
VIRTUAL ADDRESS = 29260 PHYSICAL ADDRESS = 17996
VIRTUAL ADDRESS = 56295 PHYSICAL ADDRESS = 18407
VIRTUAL ADDRESS = 58130 PHYSICAL ADDRESS = 18450
VIRTUAL ADDRESS = 4495  PHYSICAL ADDRESS = 18831
```



```
VIRTUAL ADDRESS = 16776 PHYSICAL ADDRESS = 48520
VIRTUAL ADDRESS = 11876 PHYSICAL ADDRESS = 55652
VIRTUAL ADDRESS = 26576 PHYSICAL ADDRESS = 23248
VIRTUAL ADDRESS = 48011 PHYSICAL ADDRESS = 57739
VIRTUAL ADDRESS = 1111  PHYSICAL ADDRESS = 50007
VIRTUAL ADDRESS = 10741 PHYSICAL ADDRESS = 17909
VIRTUAL ADDRESS = 12576 PHYSICAL ADDRESS = 35872
VIRTUAL ADDRESS = 24476 PHYSICAL ADDRESS = 6812
VIRTUAL ADDRESS = 54311 PHYSICAL ADDRESS = 21031
VIRTUAL ADDRESS = 47746 PHYSICAL ADDRESS = 63362
VIRTUAL ADDRESS = 19311 PHYSICAL ADDRESS = 63599
VIRTUAL ADDRESS = 21676 PHYSICAL ADDRESS = 15532
VIRTUAL ADDRESS = 62712 PHYSICAL ADDRESS = 43768
VIRTUAL ADDRESS = 5141  PHYSICAL ADDRESS = 47893
VIRTUAL ADDRESS = 29376 PHYSICAL ADDRESS = 18112
VIRTUAL ADDRESS = 39611 PHYSICAL ADDRESS = 24507
VIRTUAL ADDRESS = 26311 PHYSICAL ADDRESS = 22727
VIRTUAL ADDRESS = 676   PHYSICAL ADDRESS = 43172
VIRTUAL ADDRESS = 60177 PHYSICAL ADDRESS = 39185
VIRTUAL ADDRESS = 60877 PHYSICAL ADDRESS = 32717
VIRTUAL ADDRESS = 58777 PHYSICAL ADDRESS = 52121
VIRTUAL ADDRESS = 65077 PHYSICAL ADDRESS = 42293
VIRTUAL ADDRESS = 46176 PHYSICAL ADDRESS = 63840
VIRTUAL ADDRESS = 54746 PHYSICAL ADDRESS = 41178
VIRTUAL ADDRESS = 63847 PHYSICAL ADDRESS = 25191
VIRTUAL ADDRESS = 19141 PHYSICAL ADDRESS = 39877
VIRTUAL ADDRESS = 52911 PHYSICAL ADDRESS = 13487
VIRTUAL ADDRESS = 51946 PHYSICAL ADDRESS = 49386
VIRTUAL ADDRESS = 6711  PHYSICAL ADDRESS = 54327
VIRTUAL ADDRESS = 59477 PHYSICAL ADDRESS = 37205
VIRTUAL ADDRESS = 62977 PHYSICAL ADDRESS = 45057
VIRTUAL ADDRESS = 52477 PHYSICAL ADDRESS = 56829
VIRTUAL ADDRESS = 18441 PHYSICAL ADDRESS = 29449
VIRTUAL ADDRESS = 55012 PHYSICAL ADDRESS = 16868
VIRTUAL ADDRESS = 28241 PHYSICAL ADDRESS = 28241

TLB HIT RATE = 1.30 %
TLB MISS RATE = 98.70 %
PAGE TABLE HIT RATE = 74.90 %
PAGE TABLE MISS RATE = 25.10 %
```

APPENDIX

Source Code

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


const int VM_SIZE = 256;
const int PAGE_SIZE = 256;

const int TLB_SIZE = 16;

const int MM_SIZE = 256;


#define MAX_LINE_LENGTH 1024


int main(int argc, char* argv[]) {

    FILE *fd;


    if (argc < 2) {

        printf("NOT ENOUGH ARGUMENTS\nEXITING\n");

        return 0;

    }
```

```
fd = fopen("address.txt", "r");
if (fd == NULL) {
    printf("ERROR OPENING FILE\nFILE FAILED TO OPEN\n");
    return 0;
}

char value[MAX_LINE_LENGTH];
long long page_no, offset, totalhits = 0, faults = 0, pages = 0;

int qp = 0; // to maintain the queue position
int physicalad = 0, frame, logicalad;

int tlb[TLB_SIZE][2];
int pagetable[PAGE_SIZE];

memset(tlb, -1, sizeof(tlb));
memset(pagetable, -1, sizeof(pagetable));

while (fgets(value, sizeof(value), fd) != NULL) {
    pages++;
    logicalad = atoi(value);
```

```
// Get page number and offset from logical address
page_no = (logicalad >> 8) & 0xFF;  // Masking for page number
offset = logicalad & 0xFF;          // Masking for offset


int hit = 0; // 1 if found in TLB


// CHECK IN TLB
for (int i = 0; i < TLB_SIZE; i++) {
    if (tlb[i][0] == page_no) {
        hit = 1;
        totalhits++;
        frame = tlb[i][1];
        break;
    }
}


// If present in TLB
if (hit) {
    printf("TLB HIT\n");
} else {
```

```
// Search in page table
int f = 0;
for (int i = 0; i < PAGE_SIZE; i++) {
    if (pagetable[i] == page_no) {
        frame = i;
        f = 1;
        break;
    }
    if (pagetable[i] == -1) {
        pagetable[i] = page_no;
        frame = i;
        f = 1;
        faults++;
        break;
    }
}
// If page not found, we should replace it (not shown in your code)
if (!f) {
    // Implement page replacement logic if needed
    faults++;
    frame = 0; // Example: replace with frame 0 (this logic should be
improved)
```

```
            }

            // Replace in TLB using FIFO
            tlb[qp][0] = page_no;
            tlb[qp][1] = frame;
            qp = (qp + 1) % TLB_SIZE; // Wrap around correctly
        }

        physicalad = frame * PAGE_SIZE + offset;
        printf("VIRTUAL ADDRESS = %d\tPHYSICAL ADDRESS =
%d\n", logicalad, physicalad);
    }

    fclose(fd);  // Close the file


    double hitrate = (double)totalhits / pages * 100;
    double faultrate = (double)faults / pages * 100;

    printf("\nTLB HIT RATE = %.2f %%\n", hitrate);
    printf("TLB MISS RATE = %.2f %%\n", (100 - hitrate));
    printf("PAGE TABLE HIT RATE = %.2f %%\n", (1 - (faultrate / 100)) *
100);
```

```
    printf("PAGE TABLE MISS RATE = %.2f %%\n", faultrate);


    return 0;

}
```

CHAPTER 5

CONCLUSION

In conclusion, this program effectively demonstrates the core concepts of virtual memory management, including page tables, demand paging, and page replacement strategies. By simulating a virtual memory system, it illustrates how operating systems handle memory allocation, allowing processes to operate seamlessly beyond physical memory limits. The implementation of a FIFO page replacement algorithm ensures efficient memory usage while providing a user-friendly interface for interaction. This program serves as a valuable educational tool, enabling users to understand the complexities of memory management through practical experience. Ultimately, it emphasizes the significance of virtual memory in enhancing system performance, multitasking capabilities, and process isolation, which are fundamental to modern computing environments.