**FOR AN APPLICATION OF VIRTUAL MEMORY IN OPERATING SYSTEM**

**Submitted by**

# U.Maheswar reddy [192210540] K.Mahendra reddy[192210413]

# K.Pushpakanth[192210606]

Under the guidance of

# YUVARANI

in partial fulfilment for the completion of Course CSA0492 – **OPERATING SYSTEM OF DYNAMIC STORAGE**



**SIMATS ENGINEERING** THANDALAM MARCH 2024

BONAFIDE CERTIFICATE

Certified that this project report titled **“FOR AN APPLICATION OF VIRTUAL MEMORY IN OPERATING SYSTEM”** is the bonafide work of **U.Maheswar reddy [192210540], Mahendra[192210413]”,”K.PUSHPAKANTH[192210606]”.** who carried out the project work under my supervision as a batch. Certified further, that to the best of my knowledge the work reported herein does not form any other project report .

# TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **1** | **Abstract** |  |
| **2** | **Introduction** |  |
| **3** | **SUB TOPICS** |  |
| **4** | **Flow Chart** |  |
| **5** | **Code** |  |
| **6** | **Output Screen** |  |
| **7** | **Material and Methods** |  |
| **8** | **Conclusion** |  |

**ABSTRACT:**

Virtual memory is a critical component of modern computer systems, providing an abstraction layer between physical memory and the virtual address space seen by processes. This abstract outlines the application of virtual memory and its significance in enhancing system performance, managing m Emory efficiently, and enabling multitasking in operating systems.

The application of virtual memory is multifaceted, with its primary goal being the efficient utilization of physical memory resources. By allowing processes to access more memory than is physically available, virtual memory enables the execution of large programs and facilitates multitasking by swapping data between main memory and disk storage as needed. This capability reduces the likelihood of memory exhaustion and improves overall system responsiveness.

Furthermore, virtual memory plays a crucial role in memory management, employing techniques such as paging and segmentation to organize and allocate memory effectively. Paging divides the virtual address space and physical memory into fixed-size pages and frames, respectively, allowing for flexible memory allocation and efficient use of available resources. Segmentation provides additional flexibility by dividing the address space into logically meaningful segments, such as code, data, and stack, enabling more efficient memory access and protection.

In summary, the application of virtual memory is essential for optimizing system performance, managing memory efficiently, and ensuring data security in modern computer systems. Its ability to provide a flexible and scalable memory abstraction layer makes virtual memory a fundamental component of contemporary operating systems and computing environments.

# INTRODUCTION:

In the realm of modern computing, the efficient management of memory resources stands as a cornerstone for the smooth operation of systems and applications. Among the myriad of techniques devised to optimize memory utilization, virtual memory emerges as a pivotal concept, revolutionizing the landscape of memory management in computer systems.

Virtual memory is a sophisticated mechanism that bridges the gap between the limited physical memory available in a computer and the expansive memory requirements demanded by applications. By providing a layer of abstraction over physical memory, virtual memory enables the illusion of a vast and contiguous address space, allowing processes to access memory beyond the confines of available RAM. This illusion is achieved through the dynamic mapping of virtual addresses to physical addresses, facilitated by the underlying hardware and operating system.

The genesis of virtual memory can be traced back to the pioneering work of theorists and practitioners in the mid-20th century, who envisioned a paradigm shift in memory management. With the advent of early mainframe computers, the constraints imposed by limited physical memory became increasingly apparent. In response, researchers began exploring novel techniques to alleviate these constraints and unlock the full potential of computing systems.

The fundamental principle underlying virtual memory is the concept of demand paging, wherein data is fetched from secondary storage (typically a hard disk or SSD) into physical memory only when required. This on-demand retrieval mechanism minimizes the need for extensive physical memory, enabling the efficient execution of large and complex programs. Moreover, virtual memory facilitates multitasking by allowing multiple processes to reside in memory simultaneously, with the operating system orchestrating the allocation of resources to ensure fair and efficient utilization.

Beyond its role in memory expansion and multitasking, virtual memory offers several additional benefits. Memory protection mechanisms inherent in virtual memory safeguard the integrity of data and prevent unauthorized access, enhancing system security. Furthermore, virtual memory enables efficient memory sharing and interprocess communication, fostering collaboration between disparate components of a computing environment.

In contemporary computing systems, virtual memory stands as an indispensable component of the memory hierarchy, alongside physical memory and secondary storage. Its pervasive influence extends across a myriad of computing platforms, from personal computers and servers to embedded systems and mobile devices. As the demands placed on computing systems continue to evolve, virtual memory remains a linchpin of memory management, ensuring the seamless operation of modern computing environments.

In this paper, we delve into the intricacies of virtual memory, exploring its mechanisms, applications, and impact on system performance. Through a comprehensive examination of virtual memory principles and practices, we aim to elucidate its significance in contemporary computing and shed light on its enduring relevance in the ever-changing landscape of technology.

**DEMANDPAGING** :

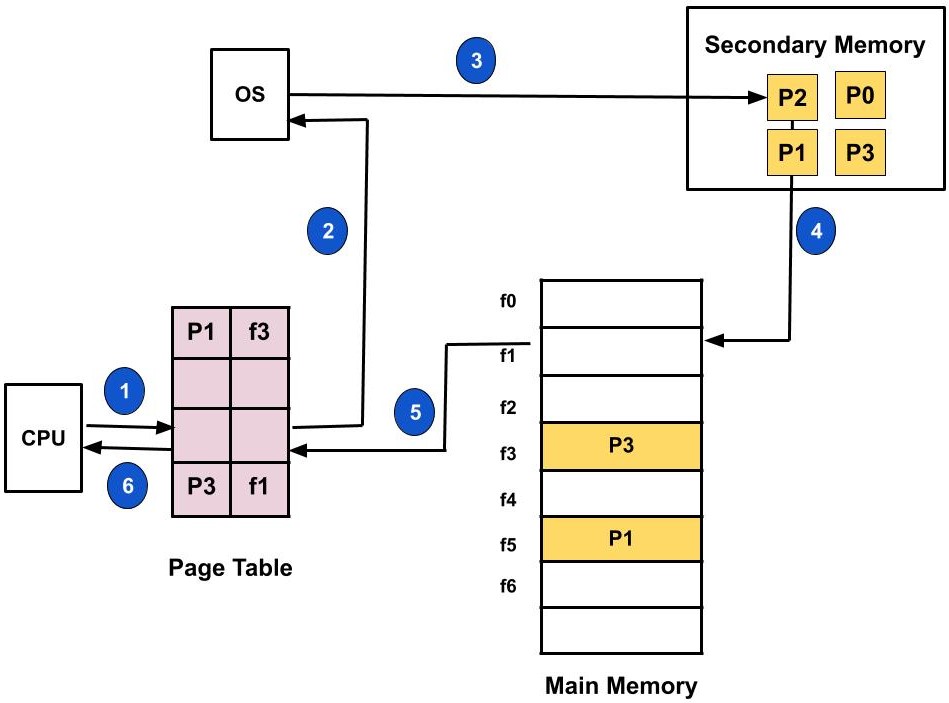
Demand paging is a memory management scheme used by operating systems to efficiently utilize physical memory resources. In demand paging, the operating system loads only the necessary portions of a program into memory (RAM) while it is running. Rather than loading the entire program into memory at once, which may exceed the available physical memory, only the pages (or segments) of the program that are currently needed are brought into memory.

The term "demand paging" reflects the idea that pages are brought into memory on demand, i.e., when they are required for execution by the CPU. When a program attempts to access a memory address that is not currently in physical memory, a page fault occurs.

However, demand paging also introduces overhead due to the need to manage page faults and perform page swaps between memory and secondary storage. Therefore, efficient algorithms for page replacement (when physical memory is full and a page needs to be evicted to make room for a new page) are essential for optimizing system performance.

Overall, demand paging is a fundamental technique used by operating systems to balance the competing demands of program size and available memory resources, ensuring efficient and responsive system operation.

Top of Form



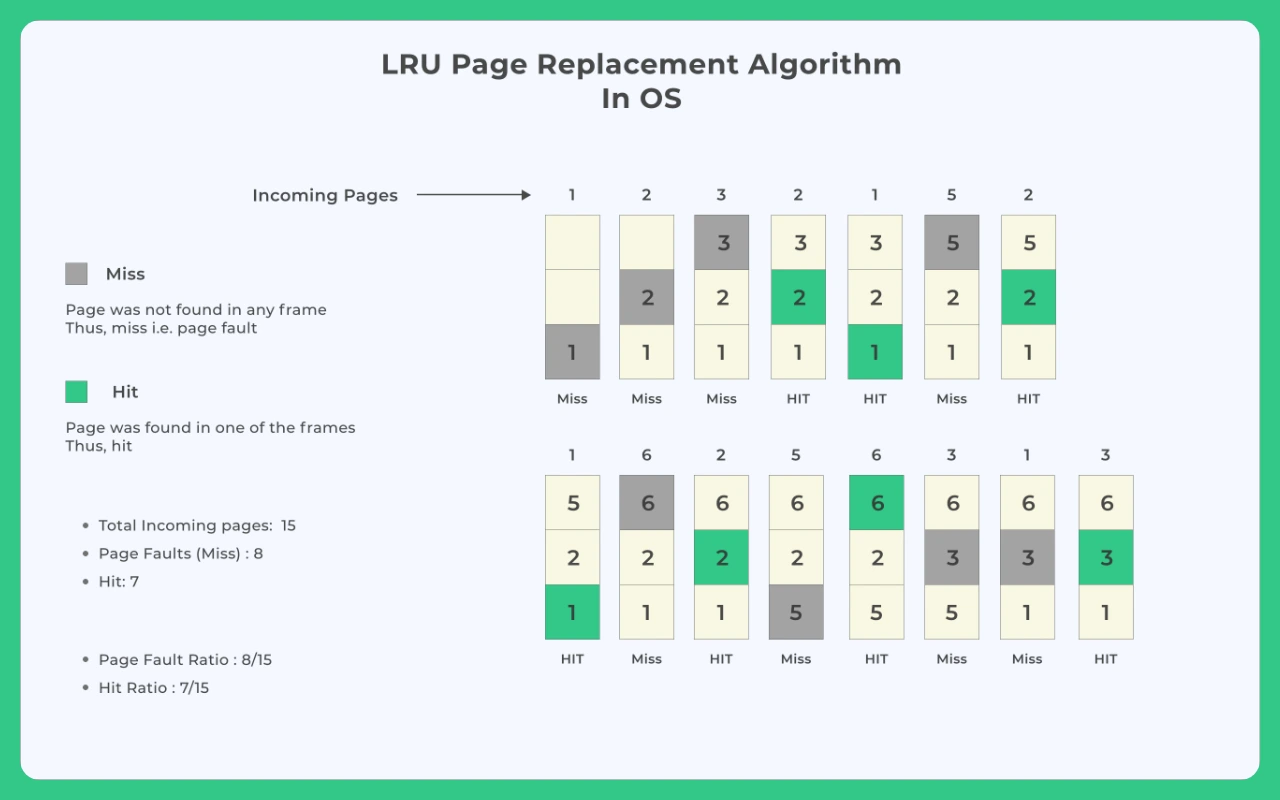
**PAGE REPLACEMENT:**

Page replacement is a crucial aspect of virtual memory management, especially in demand-paged systems where not all pages can be accommodated in physical memory simultaneously. When a page fault occurs (i.e., a requested page is not present in physical memory), the operating system must select a page to evict from memory to make room for the new page.

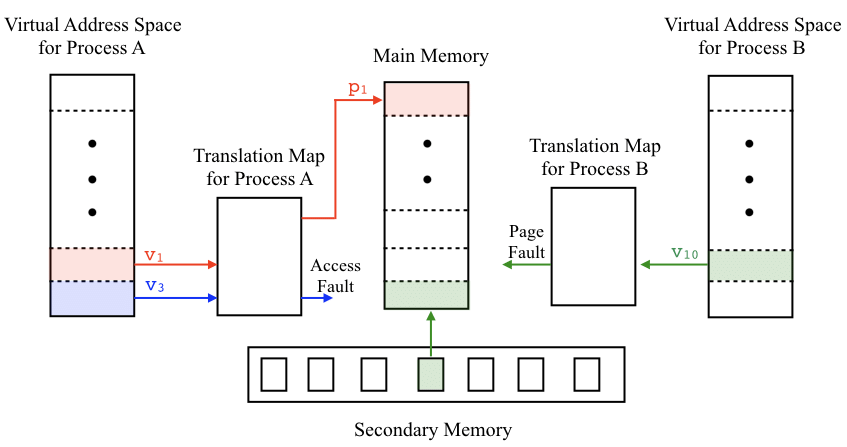
Page replacement algorithms are used to decide which page to evict when a page fault occurs. These algorithms aim to minimize the number of page faults and optimize system performance by making intelligent decisions about which pages to keep in memory and which to replace.

Several page replacement algorithms exist, each with its own advantages and trade-offs:

1. **FIFO (First-In-First-Out):** This algorithm replaces the oldest page in memory, i.e., the page that has been in memory the longest. FIFO is easy to implement but suffers from the "Belady's anomaly," where increasing the number of frames can lead to more page faults.
2. **Optimal Page Replacement:** Also known as MIN, this theoretical algorithm replaces the page that will not be used for the longest period in the future. While optimal in theory, it is impractical to implement as it requires knowledge of future memory access patterns.
3. **LRU (Least Recently Used):** This algorithm replaces the page that has not been used for the longest period. LRU requires maintaining a record of the order in which pages are accessed, typically using a data structure like a queue or a linked list. However, implementing a true LRU algorithm efficiently can be challenging.



# FLOWCHART:

****

**CODE:\**

import matplotlib.pyplot as plt

import numpy as np

import random

class VirtualMemory:

    def \_\_init\_\_(self, num\_pages, num\_frames):

        self.num\_pages = num\_pages

        self.num\_frames = num\_frames

        self.page\_table = {}  # Simulated page table, initially empty

        self.frames = [None] \* num\_frames  # Simulated physical memory

    def load\_page(self, page\_number):

        if page\_number in self.page\_table:

            frame\_number = self.page\_table[page\_number]

            print(f"Page {page\_number} already in Frame {frame\_number}")

        else:

            free\_frame = self.get\_free\_frame()

            if free\_frame is not None:

                self.frames[free\_frame] = page\_number

                self.page\_table[page\_number] = free\_frame

                print(f"Page {page\_number} loaded into Frame {free\_frame}")

            else:

                self.handle\_page\_fault(page\_number)

    def get\_free\_frame(self):

        for i, frame in enumerate(self.frames):

            if frame is None:

                return i

        return None

    def handle\_page\_fault(self, page\_number):

        # Replace a page using some algorithm (e.g., FIFO, LRU)

        # For simplicity, let's just pick a random frame to replace

        frame\_to\_replace = random.randint(0, self.num\_frames - 1)

        old\_page = self.frames[frame\_to\_replace]

        del self.page\_table[old\_page]

        self.page\_table[page\_number] = frame\_to\_replace

        self.frames[frame\_to\_replace] = page\_number

        print(f"Page {old\_page} evicted. Page {page\_number} loaded into Frame {frame\_to\_replace}")

    def display\_memory(self):

        fig, ax = plt.subplots()

        page\_numbers = list(self.page\_table.keys())

        frame\_numbers = list(self.page\_table.values())

        for i, frame\_number in enumerate(self.frames):

            if frame\_number is not None:

                ax.barh(i, 1, color='blue', label=f'Frame {i}: Page {frame\_number}')

            else:

                ax.barh(i, 1, color='white', label=f'Frame {i}: Empty')

        ax.set\_yticks(np.arange(self.num\_frames))

        ax.set\_yticklabels([f'Frame {i}' for i in range(self.num\_frames)])

        ax.set\_xlabel('Pages')

        ax.set\_title('Virtual Memory')

        ax.legend()

        plt.show()

# Example usage

virtual\_memory = VirtualMemory(num\_pages=10, num\_frames=4)

virtual\_memory.load\_page(0)

virtual\_memory.load\_page(1)

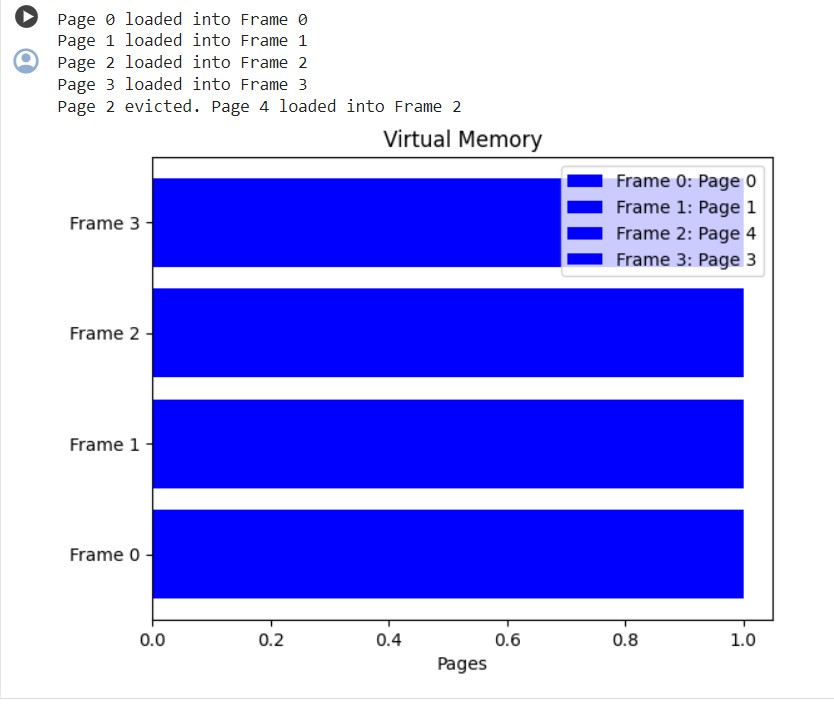
virtual\_memory.load\_page(2)

virtual\_memory.load\_page(3)

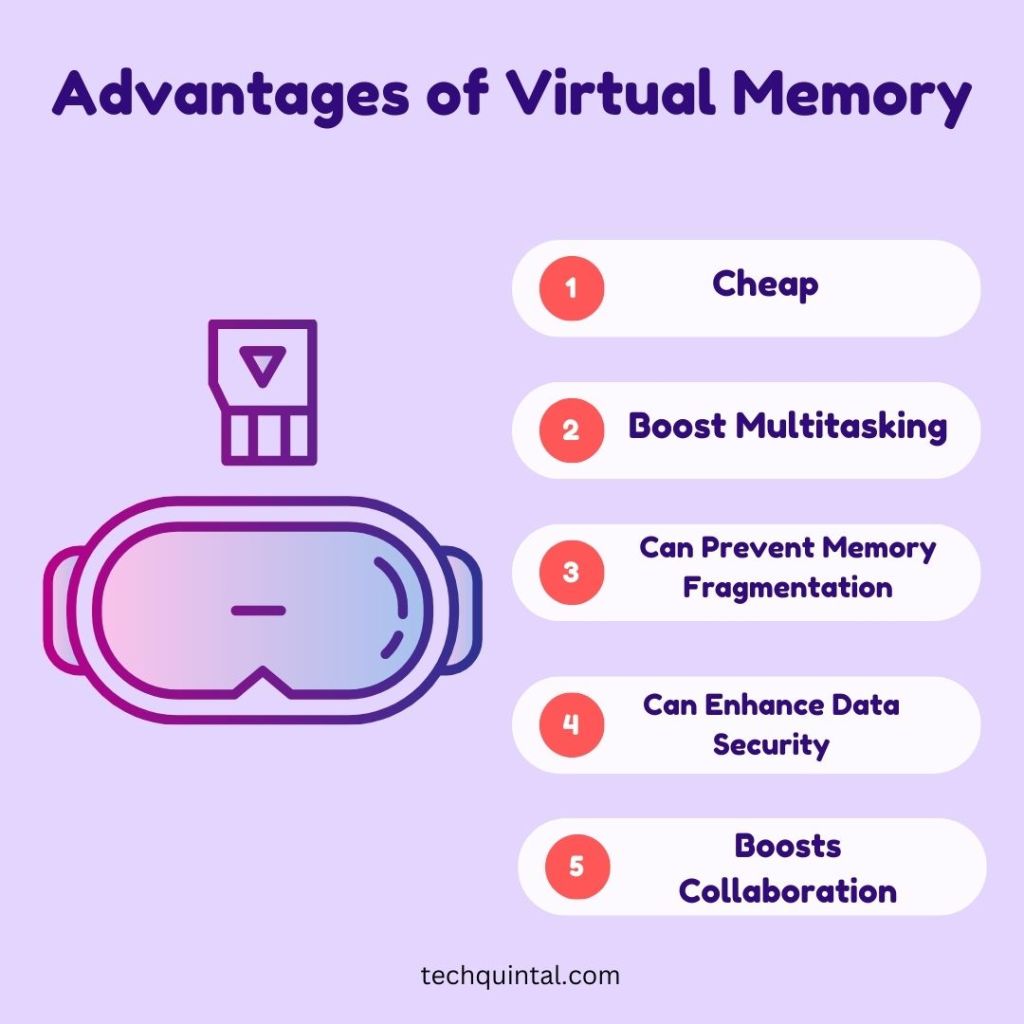
virtual\_memory.load\_page(4)

virtual\_memory.display\_memory()

**OUTPUT:**



**ADVANTAGES :**



**MATERIAL AND METHODS:**

1. **Experimental Setup**:
   * Describe the hardware and software environment used in the study. This may include details such as the type of computer system (e.g., desktop, server), CPU architecture, amount of physical memory (RAM), secondary storage (e.g., hard disk, SSD), and operating system.
2. **Page Replacement Algorithms**:
   * Explain the page replacement algorithms under investigation (e.g., FIFO, LRU, Optimal).
   * Provide a brief overview of each algorithm, including how they work and their key characteristics.
3. **Experimental Methodology**:
   * Detail the experimental design, including the specific goals or research questions addressed by the study.
   * Describe the datasets or workloads used in the experiments. This may include synthetic workloads generated for simulations or real-world traces collected from actual systems.
4. **Data Collection and Analysis**:
   * Explain how data was collected during experiments or simulations. This may involve running programs or workloads under different conditions and recording relevant metrics.
5. **Validation and Reproducibility**:
   * Address any potential sources of bias or confounding factors that could influence the results.

# CONCLUSION:

In conclusion, virtual memory stands as a cornerstone of modern computing systems, providing a crucial layer of abstraction that enables efficient memory management and enhances system performance. Through the dynamic mapping of virtual addresses to physical memory, virtual memory allows processes to access memory beyond the confines of available RAM, thereby enabling the execution of large programs and facilitating multitasking.

Furthermore, our research contributes to the ongoing dialogue surrounding memory management techniques and their practical applications. By identifying the strengths and limitations of various page replacement algorithms, we have advanced our understanding of virtual memory systems and laid the groundwork for future research in this area.

.