

VIRTUAL MEMORY MANAGER

A PROJECT REPORT

Submitted by

SRIJONI CHOWDHURY [Reg No: RA2111026010210]

SAYAL SINGH [Reg No: RA211026010218]

RIZVI [Reg No: RA211026010219]

Under the Guidance of

DR. Gopirajan PV

Assistant Professor, Department of Computational Intelligence

In partial fulfilment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

**with a specialization in Artificial Intelligence and
Machine Learning**



DEPARTMENT OF COMPUTATIONAL INTELLIGENCE

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

November 2023



SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this B.Tech project report titled “**VIRTUAL MEMORY MANAGER**” is the bonafide work of Ms. Srijoni Chowdhury [Reg. No.: RA2211026010210] , Mr. Sayal Singh [Reg. No. RA211026010218] , and Rizvi [Reg. No.: RA2211026010210] who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion for this or any other candidate.

DR. Gopirajan PV

SUPERVISOR

Assistant Professor

Department of Computational Intelligence

DR. Annie Uthra R

HEAD OF THE DEPARTMENT

Department of Computational Intelligence

**SIGNATURE OF INTERNAL
EXAMINER**

**SIGNATURE OF
EXTERNAL EXAMINER**



SRM Institute of Science and Technology

Own Work Declaration Form

Degree/ Course : B.Tech in Computer Science and Engineering with a
specialization in AI ML

Student Names : Srijoni Chowdhury, Sayal Singh, Rizvi

Registration Number: RA2211026010210, RA2211026010218, RA211026010219

Title of Work : Virtual Memory Manager

We hereby certify that this assessment compiles with the University's Rules and Regulations relating to Academic misconduct and plagiarism, as listed in the University Website, Regulations, and the Education Committee guidelines.

We confirm that all the work contained in this assessment is our own except where indicated, and that we have met the following conditions:

- Clearly references / listed all sources as appropriate
- Referenced and put in inverted commas all quoted text (from books, web, etc.)
- Given the sources of all pictures, data etc. that are not my own
- Not made any use of the report(s) or essay(s) of any other student(s) either past or present
- Acknowledged in appropriate places any help that I have received from others

(e.g. fellow students, technicians, statisticians, external sources) • Compiled with any other plagiarism criteria specified in the Course handbook / University website

I understand that any false claim for this work will be penalized in accordance with the University policies and regulations.

DECLARATION:
I am aware of and understand the University's policy on Academic misconduct and plagiarism and I certify that this assessment is my / our own work, except where indicated by referring, and that I have followed the good academic practices noted above.
If you are working in a group, please write your registration numbers and sign with the date for every student in your group.

v

ACKNOWLEDGEMENT

We express our humble gratitude to **Dr. C. Muthamizhchelvan**, Vice-Chancellor, SRM Institute of Science and Technology, for the facilities extended for the project work and his continued support.

We extend our sincere thanks to Dean-CET, SRM Institute of Science and Technology, **Dr. T.V.Gopal**, for his invaluable support.

We wish to thank **Dr. Revathi Venkataraman**, Professor & Chairperson, School of Computing, SRM Institute of Science and Technology, for her support throughout the project work.

We are incredibly grateful to our Head of the Department, **Dr Annie Uthra R**, Professor, Department of Computational Intelligence, SRM Institute of Science and Technology, for her suggestions and encouragement at all the stages of the project work.

We want to convey our thanks to our Project guide **Dr Gopirajan PV**, Assistant Professor, Department of Computational Intelligence, SRM Institute of Science and Technology, for their inputs during the project reviews and support.

vi

We sincerely thank the Computational Intelligence Department staff and students, SRM Institute of Science and Technology, for their help during our project. Finally, we would like to thank parents, family members, and friends for their unconditional love, constant support, and encouragement.

vii

TABLE OF CONTENTS

S No.	Title	Page No.
1.	Abstract	
2.	Introduction	
3.	Literature Survey	
4.	Proposed Methodology	
5.	Result	
6.	Conclusion	
7.	Future Scope	
8.	References	
9.	Appendix	
10.	Output Module	

ABSTRACT

Virtual memory plays an important role in a modern-day OS. Virtual memory is an integral part of a modern computer architecture; implementations usually require hardware support, typically in the form of a memory management unit built into the CPU. While not necessary, emulators and virtual machines can employ hardware support to increase performance of their virtual memory implementations. Older operating systems, such as those for the mainframes of the 1960s, and those for personal computers of the early to mid-1980s (e.g., DOS), generally have no virtual memory functionality. The goal of this project is to create a software that converts logical addresses to physical addresses for a virtual address space of $2^{16} = 65,536$ bytes. The application will read from a file containing logical addresses, translate each logical address to its matching physical address using a TLB and a page table, then output the value of the byte stored at the translated physical location. This project's objective is to emulate the stages involved in converting logical to physical addresses.

CHAPTER 1

INTRODUCTION

1.1.General

In the 1950s, all larger programs had to contain logic for managing primary and secondary storage, such as overlaying. Virtual memory was therefore introduced not only to extend primary memory, but to make such an extension as easy as possible for programmers to use.^[8] To allow for multiprogramming and multitasking, many early systems divided memory between multiple programs without virtual memory, such as early models of the PDP-10 via registers.

A claim that the concept of virtual memory was first developed by German physicist Fritz-Rudolf Güntsch at the Technische Universität Berlin in 1956 in his doctoral thesis, Logical Design of a Digital Computer with Multiple Asynchronous Rotating Drums and Automatic High Speed Memory Operation^{[9][10]} does not stand up to careful scrutiny. The computer proposed by Güntsch (but never built) had an address space of 10^5 words which mapped exactly onto the 10^5 words of the drums, i.e. the addresses were real addresses and there was no form of indirect mapping, a key feature of virtual memory. What Güntsch did invent was a form of cache memory, since his high-speed memory was intended to contain a copy of some blocks of code or data taken from the drums. Indeed, he wrote (as quoted in translation): "The programmer need not respect the existence of the primary memory (he need not even know that it exists), for there is only one sort of addresses (sic) by which one can program as if there were only one storage." This is exactly the situation in computers with cache memory, one of the earliest commercial examples of which was the IBM System/360 Model 85. In the Model 85 all addresses were real addresses referring to the main core store. A semiconductor cache store, invisible to the user, held the contents of parts of the main store in use by the currently executing program. This is exactly analogous to Güntsch's system,

designed as a means to improve performance, rather than to solve the problems involved in multi-programming.

IBM developed the concept of hypervisors in their CP-40 and CP-67, and in 1972 provided it for the S/370 as Virtual Machine Facility/370. IBM introduced the Start Interpretive Execution (SIE) instruction as part of 370-XA on the 3081, and VM/XA versions of VM to exploit it.

Before virtual memory could be implemented in mainstream operating systems, many problems had to be addressed. Dynamic address translation required expensive and difficult-to-build specialized hardware; initial implementations slowed down access to memory slightly.^[8] There were worries that new system-wide algorithms utilizing secondary storage would be less effective than previously used application-specific algorithms. By 1969, the debate over virtual memory for commercial computers was over;^[8] an IBM research team led by David Sayre showed that their virtual memory overlay system consistently worked better than the best manually controlled systems.^[22] Throughout the 1970s, the IBM 370 series running their virtual-storage based operating systems provided a means for business users to migrate multiple older systems into fewer, more powerful, mainframes that had improved price/performance. The first minicomputer to introduce virtual memory was the Norwegian NORD-1; during the 1970s, other minicomputers implemented virtual memory, notably VAX models running VMS.

Virtual memory was introduced to the x86 architecture with the protected mode of the Intel 80286 processor, but its segment swapping technique scaled poorly to larger segment sizes. The Intel 80386 introduced paging support underneath the existing segmentation layer, enabling the page fault exception to chain with other exceptions without double fault. However, loading segment descriptors was an expensive operation, causing operating system designers to rely strictly on paging rather than a combination of paging and segmentation.

1.2. Purpose

The purpose of a Virtual Memory Manager (VMM) project is to create a software component that manages the translation of logical addresses to physical addresses in a computer system. This is an essential part of modern operating systems and plays a crucial role in memory

management. Here are the key purposes of such a project:

- **Memory Abstraction:** A VMM provides an abstraction layer to applications, allowing them to work with a larger virtual address space that may exceed the available physical memory. This abstraction simplifies memory management for applications, making it easier to develop and run software.
- **Address Space Isolation:** It ensures that each running process has its isolated address space, preventing one process from accessing the memory of another. This is a fundamental aspect of process separation and security.
- **Efficient Use of Physical Memory:** The VMM optimizes the use of physical memory by swapping data in and out of RAM as needed. It allows for running more processes simultaneously than the available physical memory would otherwise allow.
- **Demand Paging:** A VMM can implement demand paging, where only the necessary portions of a program are loaded into physical memory when needed. This minimizes the initial memory footprint of a program and reduces load times.
- **Page Table Management:** The project involves managing page tables, which are data structures used for translating logical addresses to physical addresses. Efficient algorithms and data structures are needed for this purpose.
- **Fault Handling:** When a program accesses a part of the address space that is not in physical memory (a page fault), the VMM must handle this situation, loading the required data from secondary storage (e.g., a hard disk) into memory.

- **Memory Protection:** The VMM enforces memory protection, preventing unauthorized access to memory locations. It ensures that processes can only access their own memory.
- **Swapping:** The VMM handles the swapping of pages between physical memory and secondary storage to ensure that the most active or relevant pages are in RAM.
- **Resource Management:** The VMM is responsible for managing limited system resources, such as physical memory and disk space, to ensure that various processes can execute smoothly.
- **Performance Optimization:** Optimizing the VMM for performance is critical to ensure that the overhead introduced by virtual memory management is minimal, and applications can run efficiently.

A Virtual Memory Manager project is essential for enhancing system performance, resource management, and providing a convenient and secure abstraction of memory for applications running in a computer system. It plays a vital role in ensuring that multiple processes can run concurrently while effectively utilizing the available hardware resources.

1.3 Scope

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size $2^{16} = 65,536$ bytes. Your program will read from a file containing logical addresses and, using a TLB as well as a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address.

VMMs have a significant and enduring presence in the IT industry, and their scope and compasses various areas and trends:

- **Data Centers and Cloud Computing:** VMMs are a fundamental component of modern data center and cloud infrastructure, enabling server virtualization and efficient resource management. As cloud computing continues to grow, VMMs are essential for delivering scalable and cost-effective cloud services.
- **Server Virtualization:** VMMs continue to play a crucial role in server consolidation, resource optimization, and workload management. They enable organizations to maximize the utilization of physical hardware, reducing the need for excessive physical servers.
- **Edge Computing:** VMMs are becoming increasingly relevant in edge computing environments, where virtualization helps in managing and isolating workloads on edge devices.
- **Security:** VMMs are used in the field of cybersecurity for creating isolated environments to analyze and contain threats, such as malware and advanced persistent threats (APTs).
- **IoT (Internet of Things):** As IoT devices become more prevalent, VMMs can be used to create secure and isolated environments for running IoT applications and services.
- **DevOps and Continuous Integration/Continuous Deployment (CI/CD):** VMMs are incorporated into CI/CD pipelines to automate the testing and deployment of software applications in isolated environments.
- **Resource Management:** VMMs provide sophisticated resource management features, including dynamic allocation, load balancing, and QoS (Quality of Service), making them critical in resource-intensive applications.
- **Disaster Recovery:** VMMs are a key component in disaster recovery planning, offering features like VM snapshots, replication, and failover to ensure business continuity.
- **Hybrid Clouds:** VMMs are essential in hybrid cloud environments that combine on-premises infrastructure with public and private clouds, allowing for seamless workload migration and scalability.
- **Artificial Intelligence and Machine Learning:** VMMs can be used to create isolated environments for AI and ML experimentation and training, enabling researchers to work with different hardware and software configurations.
- **IoT Edge Computing:** In IoT scenarios, VMMs enable the efficient management and isolation of computer workloads at the edge, where IoT devices generate data and require processing.

.The scope of VMMs continues to expand as they adapt to emerging technologies and trends. They remain a critical component in modern computing environments, providing agility, efficiency, security, and scalability for a wide range of applications and industries.

1.4 Memory Management

Memory is central to the operation of a modern computer system and consists of a large array of bytes, each with its own address. One way to allocate an address space to each process is through the use of base and limit registers. The base register holds the smallest legal physical memory address, and the limit specifies the size of the range. Binding symbolic address references to actual physical addresses may occur during compile, load, or execution time. An address generated by the CPU is known as a logical address, which the memory management unit (MMU) translates to a physical address in memory. One approach to allocating memory is to allocate partitions of contiguous memory of varying sizes. These partitions may be allocated based on three possible strategies first fit, best fit, and worst fit. Modern operating systems use paging to manage memory. In this process, physical memory is divided into fixed-sized blocks called frames and logical memory into blocks of the same size called pages. When paging is used, a logical address is divided into two parts: a page number and a page offset. The page number serves as an index into a per process page table that contains the frame in physical memory that holds the page. The offset is the specific location in the frame being referenced. A translation look-aside buffer (TLB) is a hardware cache of the page table. Each TLB entry contains a page number and its corresponding frame. Using a TLB in address translation for paging systems involves obtaining the page number from the logical address and checking if the frame for the page is in the TLB. If it is, the frame is obtained from the TLB. If the frame is not present in the TLB, it must be retrieved from the page table. Hierarchical paging involves dividing a logical address into multiple parts, each referring to different levels of page tables. As addresses expand beyond 32 bits, the number of hierarchical levels may become large. Two strategies that address this problem are hashed page tables and inverted page tables. Swapping allows the system to move pages belonging to a process to disk to increase the degree of multiprogramming.

The Intel 32-bit architecture has two levels of page tables and supports either 4-KB or 4-MB page sizes. This architecture also supports pageaddress extension, which allows 32-bit processors to access a physical address space larger than 4 GB. The x86-64 and ARMv9 architectures are 64-bit architectures that use hierarchical paging.

1.5 Virtual Memory

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the programmer from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files and libraries, and to implement shared memory. In addition, it provides an efficient mechanism for process creation. Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly.

CHAPTER 2

LITERATURE REVIEW

In this section, we discuss the previous research that has been conducted in Virtual Memory Management..

2.1. Overview on Virtual Memory

Memory-management algorithms necessary because of one basic requirement: the instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory.

Dynamic linking can help to ease this restriction, but it generally requires special precautions and extra work by the programmer. The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in many years. Even in those cases where the entire program is needed, it may not all be needed at the same time.

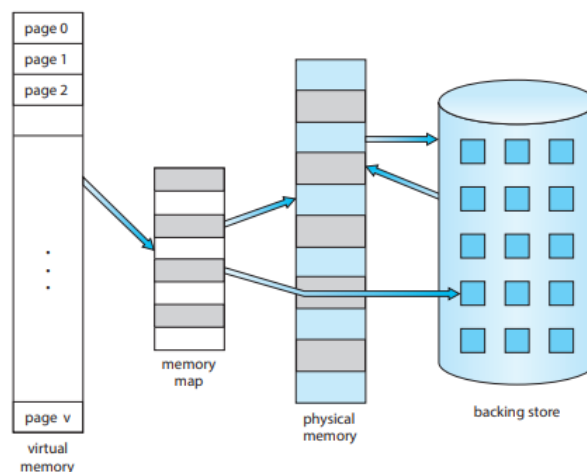
The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.
- Because each program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and its users. Virtual memory involves the separation of logical memory as perceived by developers from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 10.1).

Virtual memory makes the task of programming much easier because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on programming the problem that is to be solved. The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address—say, address 0—and exists in contiguous memory,

It is up to the memory-management unit (MMU) to map logical pages to physical page frames in memory. We allow the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow



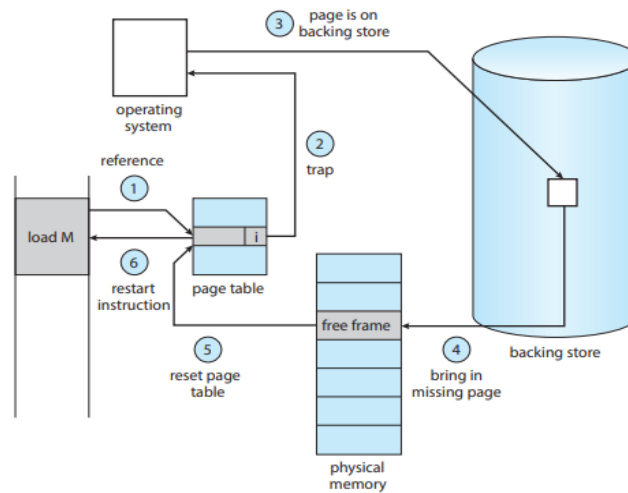
or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

Diagram showing virtual memory is larger than physical memory

2.2 Demand Paging

An executable program might be loaded from secondary storage into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether or not an option is ultimately selected by the user. An alternative strategy is to load pages only as they are needed. This technique is known as demand paging and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are thus never loaded into physical memory. A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually an HDD or NVM device). Demand paging explains one of the primary benefits of virtual memory—by loading only the portions of programs that are needed, memory is used more efficiently .

The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid. But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page fault. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward.



Steps involved in handling Page Fault

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example)
4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is pure demand paging: never bring a page into memory until it is required. Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behavior is exceedingly unlikely. Programs tend to have locality of reference, which results in reasonable performance from demand paging. The hardware to support demand paging is the same as the hardware for paging and swapping: • Page table. This table has the ability to mark an entry invalid through a valid –invalid bit or a

special value of protection bits.

- Secondary memory. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk or NVM device. It is known as the swap device, and the section of storage used for this purpose is known as swap space. Swap-space allocation A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible. In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand. As a worst-case example, consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C

If we fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs.

The major difficulty arises when one instruction may modify several different locations. For example, consider the IBM System 360/370 MVC (move character) instruction, which can move up to 256 bytes from one location to another (possibly overlapping) location. If either block (source or destination) straddles a page boundary, a page fault might occur after the move is partially done. In addition, if the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction. This problem can be solved in two different ways. In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place; we know that no page fault can occur, since all the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.

This is by no means the only architectural problem resulting from adding paging to an existing architecture to allow demand paging, but it illustrates some of the difficulties involved. Paging is added between the CPU and the memory in a computer system. It should be entirely transparent to a process. Thus, people often assume that paging can be added to any system. Although this assumption is true for a non-demand-paging environment, where a page fault represents a fatal error, it is not true where a page fault means only that an additional page must be brought into memory and the process restarted.

2.3 Free-Frame List

When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory. To resolve page faults, most operating systems maintain a free-frame list, a pool of free frames for satisfying such requests (Free frames must also be allocated when the stack or heap segments from a process expand.) Operating systems typically allocate free frames using a technique known as zero-fill-on-demand . Zero-fill-on-demand frames are “zeroed-out” before being allocated, thus erasing their previous contents. (Consider the potential



security implications of not clearing out the contents of a frame before reassigning it.)

List of free frames

When a system starts up, all available memory is placed on the free-frame list. As free frames are requested (for example, through demand paging), the size of the free-frame list shrinks. At some point, the list either falls to zero or falls below a certain threshold, at which point it must be repopulated.

CHAPTER 3

PROPOSED METHODOLOGY

In this section, we provide the details on the problem statement, the algorithms used to change from logical to physical addresses space of size $2^{16} = 65,536$ bytes. First of all we created a text file containing several 32-bit integers that will represent logical addresses. Then the program coded will read these logical addresses and give the output will translate and give their corresponding physical address and output the value of the signed bit at the physical address.

3.1. Dataset

The logical addresses provided the addresses.txt file, which contains integer values representing logical addresses ranging from 0 – 65,536 (the size of the virtual address space). The program will open this file, read each logical address and translate it to its corresponding physical address, and output the value of the signed byte at the physical address.

This program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the

rightmost 16 bits of each logical address. These 16 bits are divided into (1) an 8-bit page number and (2) 8-bit page offset

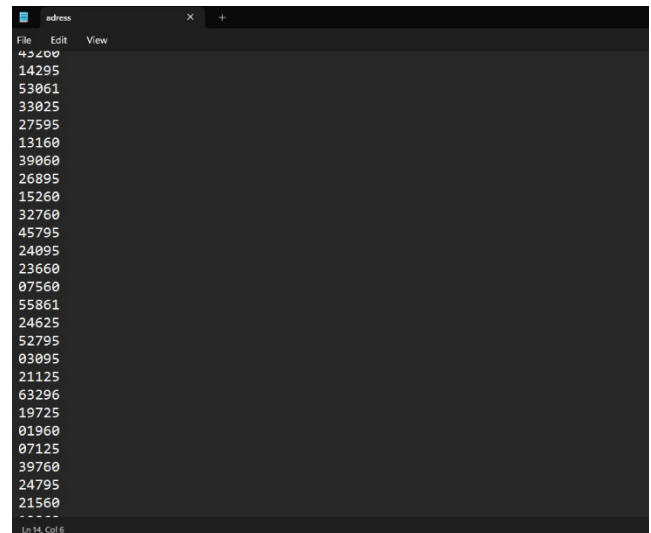
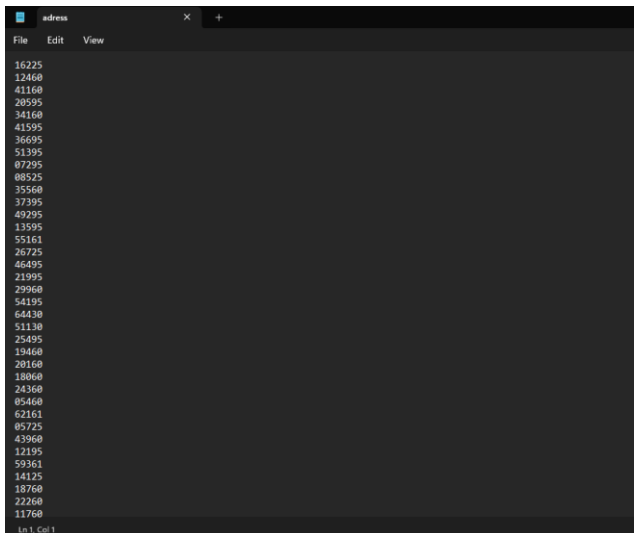
Other specifics include the following:

- 28 entries on the page table
- Page size of 28 bytes
- 16 entries in the TLB
- Frame size of 28 bytes
- 256 frames
- Physical memory of 65,536 bytes (256 frames x 256-byte frame size)

Additionally, your program need only be concerned with reading logical address and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space.

A logical address, also known as a virtual address or symbolic address, is an address that is used to access data or resources in a computer system. It is generated by the software running on the computer and is distinct from the physical address of a specific hardware component, such as a memory location or a storage device.

The purpose of using logical addresses is to provide an abstraction layer that allows software to interact with the computer's hardware without needing to know the physical details of the system.



Dataset - Logical address in address.txt

3.2. TLB Approach

A Translation Lookaside Buffer (TLB) is a hardware cache in a computer's Memory Management Unit (MMU) that plays a crucial role in the efficient translation of logical addresses to physical addresses. Here's a brief overview of TLBs and their involvement in this translation process:

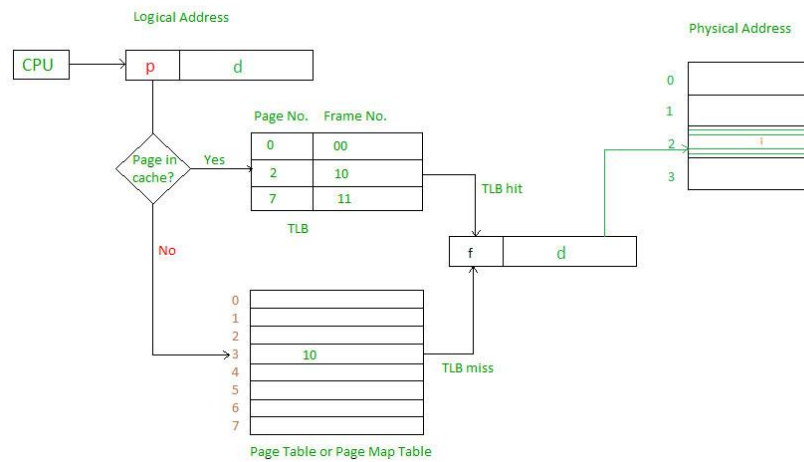
A TLB is a small, high-speed cache that stores a subset of page table entries used in address translation.. Page tables are data structures managed by the operating system, containing mappings from logical (virtual) page numbers to physical page frame numbers. TLBs are designed to speed up the translation of logical addresses to physical addresses by reducing the need to access the main memory's page table.

Address Translation Process with a TLB:

1. **Logical Address Generation:** When a program needs to access memory, it generates a logical address consisting of a page number and an offset within that page.
2. **TLB Lookup:** The MMU checks the TLB for an entry corresponding to the page number. The page number serves as an index for the TLB.

3. **TLB Hit:** If the TLB contains an entry for the page number (a TLB hit), the MMU retrieves the corresponding physical page frame number from the TLB entry. This significantly speeds up address translation, as accessing the TLB is much faster than accessing the full page table in main memory.
4. **TLB Miss:** If the TLB does not contain an entry for the page number (a TLB miss), the MMU proceeds to the next step.
5. **Page Table Lookup:** In the case of a TLB miss, the MMU looks up the page table in the main memory to find the page table entry for the given page number. This entry contains the mapping from the logical page number to the corresponding physical page frame number.
6. **TLB Update:** After fetching the page table entry during a TLB miss, the MMU typically updates the TLB with the newly retrieved information to accelerate future address translations for the same page.
7. **Address Translation:** The MMU combines the physical page frame number obtained from either the TLB or the page table with the offset from the logical address to generate the physical address.
8. **Access Memory:** With the physical address, the MMU accesses the actual memory location (e.g., RAM) to read or write data as specified by the logical address.

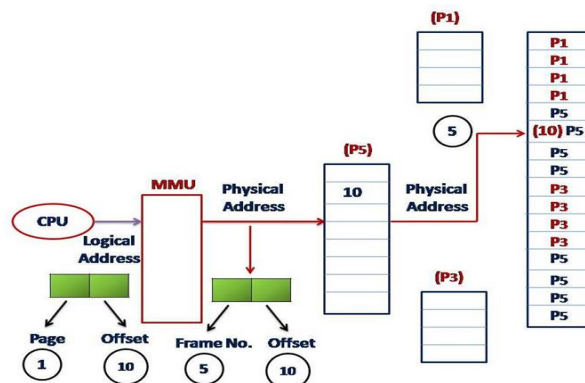
TLBs provide a significant performance benefit by reducing the number of memory accesses needed during the address translation process. The effectiveness of a TLB depends on its size, replacement policies, and how frequently page table entries are reused. Efficient TLB management is a critical aspect of modern computer architectures to optimize memory access and enhance overall system performance.



3.3 FIFO

FIFO (First-In-First-Out) is a simple and straightforward page replacement algorithm that is sometimes used in address translation, particularly for managing the contents of the TLB (Translation Lookaside Buffer). FIFO is not used for address translation itself but rather for deciding which entry in the TLB should be replaced when a new entry needs to be added due to a TLB miss. Here's how the FIFO algorithm works in the context of TLB management

:



TLB Entry Replacement: The TLB is a small, fast cache that stores page table entries to speed up the address translation process. The TLB has a limited number of entries, and when a TLB miss occurs (i.e., the desired page number is not in the TLB), a new entry needs to be added to the TLB to facilitate future translations for that page. If the TLB is already full and a replacement is necessary, the FIFO algorithm determines which entry to replace.

Replacement Decision: In a TLB managed with FIFO, the entry that has been in the TLB the longest is chosen for replacement. This means that the entry that was added first is the one that will be removed (the "first in" the TLB is the "first out").

Adding New Entry: After selecting the entry to be replaced, the new entry for the current page number is added to the TLB.

Address Translation: The address translation process then proceeds as described previously, using the new entry in the TLB for subsequent translations.

3.4 Logical Address vs Physical Address

Parameter	Logical Address	Physical Address
Basic	Generated by CPU	Location is a memory unit
Address Space	Logical Address Space is set of all logical address generated by CPU in reference to a program	Set of all physical addresses mapped to the corresponding logical address
Visibility	Visible to the user	Invisible to the user
Generation	Generated by PC	Computed by MMU
Access	User can use the logical address to access the physical address	User cannot directly access the physical address
Editable	Can be changed	Cannot be changed
Also Called	Virtual Address	Real Address

3.5 Algorithm

1. Initialization:

- Define constants for the virtual memory size (VM_SIZE), page size (PAGE_SIZE), TLB size (TLB_SIZE), and main memory size (MM_SIZE).
- Open the file specified as a command-line argument for reading addresses.
- Check for command-line argument correctness and file open success.

2. Variable Initialization:

- Initialize variables for page number (page no), offset (offset), page table (page table), total TLB hits (total hits), TLB faults (fault), total pages (pages), queue position (qp), physical address (physicalad), frame, logical address (logicalad), and mask values for bit masking.

3. Memory Structures Initialization:

- Initialize the TLB (tlb) and page table (pagetable) arrays with -1 values to represent empty entries.

4. Processing Addresses:

- Start reading addresses from the file line by line.
- For each address read:
- Increment the total pages count (pages).
- Extract the page number and offset from the logical address.
- Check if the page number is present in the TLB (TLB hit) by iterating through the TLB array. If found, set the 'hit' variable to 1 and retrieve the frame.
- If a TLB hit occurs, print "TLB HIT."
- If not found in the TLB (TLB miss):
- Iterate through the page table to check if the page number is present. If found, set the frame, and increment the TLB faults.
- If not found in the page table (page table miss), find an empty entry, or replace an entry (using FIFO) in the page table, update it, and set the frame. Then, update the TLB with the new entry.
- Print the virtual and physical addresses.

5. Hit Rate Calculation:

- Calculate the TLB hit rate (hitrate) and page table miss rate (faultrate) as percentages based on the total hits, faults, and total pages.

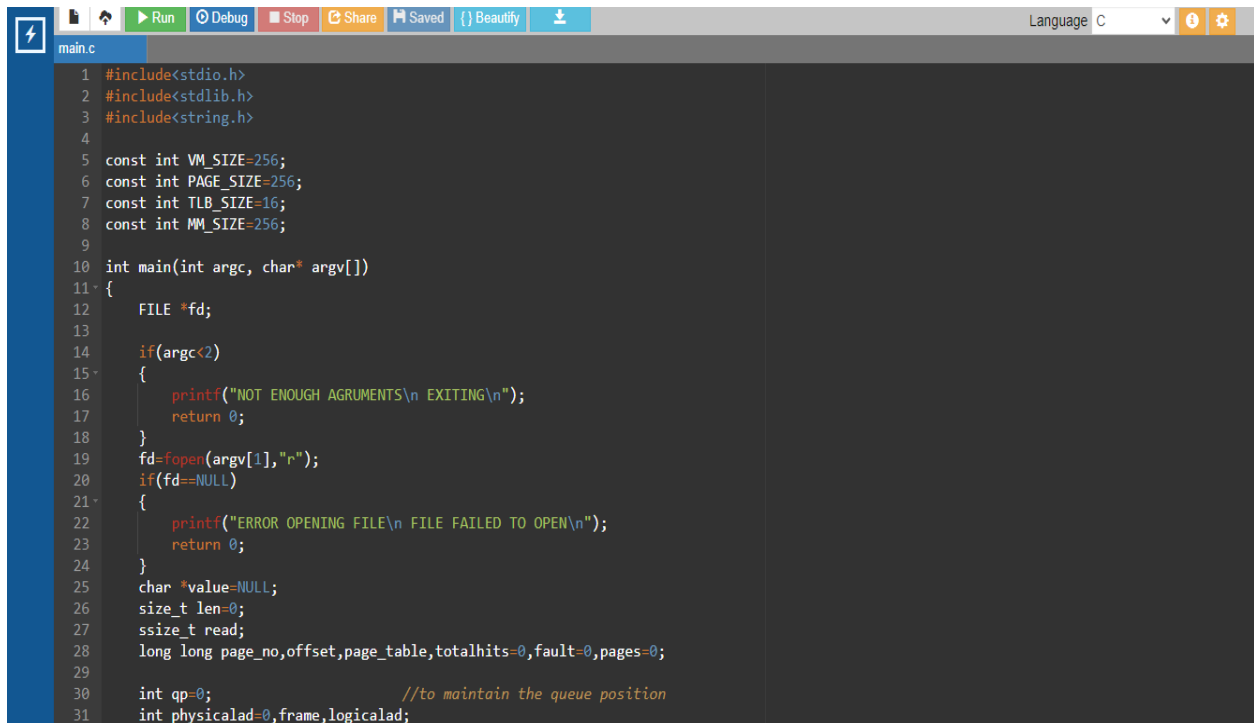
6. Printing Results:

- Print the TLB hit rate, TLB miss rate, page table hit rate, and page table miss rate.

7. Cleanup:

- Close the file and release allocated memory.

3.6 Program



```
main.c
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<string.h>
4
5 const int VM_SIZE=256;
6 const int PAGE_SIZE=256;
7 const int TLB_SIZE=16;
8 const int MM_SIZE=256;
9
10 int main(int argc, char* argv[])
11 {
12     FILE *fd;
13
14     if(argc<2)
15     {
16         printf("NOT ENOUGH AGRUMENTS\n EXITING\n");
17         return 0;
18     }
19     fd=fopen(argv[1],"r");
20     if(fd==NULL)
21     {
22         printf("ERROR OPENING FILE\n FILE FAILED TO OPEN\n");
23         return 0;
24     }
25     char *value=NULL;
26     size_t len=0;
27     ssize_t read;
28     long long page_no,offset,page_table,totalhits=0,fault=0,pages=0;
29
30     int qp=0; //to maintain the queue position
31     int physicalad=0,frame,logicalad;
```

```
main.c
29
30     int qp=0;                                //to maintain the queue position
31     int physicalad=0,frame,logicalad;
32
33     int tlb[TLB_SIZE][2];
34     int pagetable[PAGE_SIZE];
35
36     memset(tlb,-1,TLB_SIZE*2*sizeof(tlb[0][0]));
37     memset(pagetable,-1,sizeof(pagetable));
38
39     int mask=255,mask1=62580,i,hit;
40
41     while((read=getline(&value,&len,fd))!=-1)
42     {
43         pages++;
44         //get page number and offset from logical address
45         page_no=atoi(value);
46         page_no=page_no>>8;
47         page_no=page_no & mask;
48
49         offset=atoi(value);
50         offset=offset & mask;
51
52         logicalad=atoi(value);
53         //printf("%lld %lld\n",page_no,offset);
54         frame=0,physicalad=0;
55
56         hit=0;                                //1 if found in TLB
57
58         //CHECK IN TLB
59
60         for(i=0;i<TLB_SIZE;i++)
```

```
main.c
58         //CHECK IN TLB
59
60         for(i=0;i<TLB_SIZE;i++)
61         {
62             if(tlb[i][0]==page_no)
63             {
64                 hit=1;
65                 totalhits++;
66                 frame=tlb[i][1];
67                 break;
68             }
69         }
70         //if present in tlb
71         if(hit==1)
72             printf("TLB HIT\n");
73
74         //search in pagetable
75         else
76         {
77
78             int f=0;
79             for(i=0;i<PAGE_SIZE;i++)
80             {
81                 if(pagetable[i]==page_no)
82                 {
83                     frame=i;
84                     fault++;
85                     break;
86                 }
87             }
88             if(pagetable[i]==-1)
89             {
```

```
main.c
85         break;
86     }
87     if(pagetable[i]==-1)
88     {
89         f=1;
90     }
91     break;
92 }
93 }
94 if(f==1)
95 {
96     pagetable[i]=page_no;
97     frame=i;
98 }
99 //replace in tlb using fifo
100 tlb[qp][0]=page_no;
101 tlb[qp][1]=i;
102 qp++;
103 qp=qp%15;
104 }
105 if(logicalad<10000)
106 printf("VIRTUAL ADDRESS = %d \t\t\t",logicalad);
107 else
108 printf("VIRTUAL ADDRESS = %d \t\t",logicalad);
109
110 physicalad=frame*PAGE_SIZE + offset;
111 printf("PHYSICAL ADDRESS = %d\n",physicalad);
112 }
113 double hitrate=(double)totalhits/pages*100;
114 double faultrate=(double)fault/pages*100;
115 printf("\nTLB HIT RATE= %.2f %c", hitrate,'%');
116 printf("\nTLB MISS RATE= %.2f %c", (100-hitrate), '%');
```

```
main.c
93     }
94     if(f==1)
95     {
96         pagetable[i]=page_no;
97         frame=i;
98     }
99     //replace in tlb using fifo
100     tlb[qp][0]=page_no;
101     tlb[qp][1]=i;
102     qp++;
103     qp=qp%15;
104 }
105 if(logicalad<10000)
106 printf("VIRTUAL ADDRESS = %d \t\t\t",logicalad);
107 else
108 printf("VIRTUAL ADDRESS = %d \t\t",logicalad);
109
110 physicalad=frame*PAGE_SIZE + offset;
111 printf("PHYSICAL ADDRESS = %d\n",physicalad);
112 }
113 double hitrate=(double)totalhits/pages*100;
114 double faultrate=(double)fault/pages*100;
115 printf("\nTLB HIT RATE= %.2f %c", hitrate,'%');
116 printf("\nTLB MISS RATE= %.2f %c", (100-hitrate), '%');
117 printf("\nPAGE TABLE HIT RATE= %.2f %c", faultrate,'%');
118 printf("\nPAGE TABLE MISS RATE= %.2f %c\n", (100-faultrate), '%');
119 }
120
121
122
```

3.7 Running the program

- The program should run as follows: `< ./a.out addresses.txt >`
- The program will read in the file `addresses.txt`, which contains 1,000 logical addresses ranging from 0 – 65535.
- The program is to translate each logical address to a physical address and determine the contents of the signed byte stored at the correct physical address. (Hint: in the C language, the `char` data type occupies a byte of storage, so we suggest using `char` values.)
- The program is to output the following values:
- The logical address being translated (the integer value being read from `addresses.txt`).
- The corresponding physical address (what your program translates the logical address to).
- The signed byte value stored at the translated physical address

3.8 Output

```
[root@localhost ~]# gcc MANAGER.C
gcc: error: MANAGER.C: C++ compiler not installed on this system
[root@localhost ~]# gcc manager.c
[root@localhost ~]# ./a.out address.txt
VIRTUAL ADDRESS = 16225      PHYSICAL ADDRESS = 97
VIRTUAL ADDRESS = 12460      PHYSICAL ADDRESS = 428
VIRTUAL ADDRESS = 41160      PHYSICAL ADDRESS = 712
VIRTUAL ADDRESS = 20595      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 = 3705
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 = 64430      PHYSICAL ADDRESS = 5294
VIRTUAL ADDRESS = 51130      PHYSICAL ADDRESS = 5562
VIRTUAL ADDRESS = 25495      PHYSICAL ADDRESS = 5783
VIRTUAL ADDRESS = 19460      PHYSICAL ADDRESS = 5892
VIRTUAL ADDRESS = 20160      PHYSICAL ADDRESS = 6336
VIRTUAL ADDRESS = 18060      PHYSICAL ADDRESS = 6540
```



```

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 = 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= 73.60 %
PAGE TABLE MISS RATE= 26.40 %
[root@localhost ~]#

```



3.9 Result

The program has been successfully executed. The program reads the addresses from the file, performs address translation, and calculates TLB hit rates and page table miss rates based on the provided addresses.

CHAPTER 4

CONCLUSION

In this project, we set out to implement a simplified model of virtual memory management, simulating the address translation process using a page table and a Translation Lookaside Buffer (TLB). Our objective was to gain a better understanding of the key concepts involved in virtual memory and TLB usage. Through the implementation and execution of the program, several important insights and outcomes were achieved:

- Address Translation:** We successfully demonstrated the address translation process from logical addresses to physical addresses, showcasing the role of page numbers and offsets.
- TLB Usage:** We introduced the concept of the TLB as a cache for frequently used page table entries. The program effectively checked the TLB for TLB hits and misses, showing how it can significantly improve address translation performance.
- Page Table:** We created a page table that mapped virtual page numbers to physical page frame numbers. Page table entries were managed effectively, and page table hits and misses were tracked.
- Replacement Strategy:** We used a simple FIFO (First-In-First-Out) strategy for TLB entry replacement, a crucial component in managing the TLB.
- Performance Metrics:** The program provided insights into TLB hit rates, TLB miss rates, page table hit rates, and page table miss rates. This allowed us to evaluate the effectiveness of the TLB in improving address translation performance.
- Educational Value:** This project served as an educational exercise to understand the fundamentals of virtual memory, TLBs, and address translation, providing practical experience in implementing these concepts.
- Limitations:** It's important to note that this project is a simplified model for educational purposes and lacks many features found in real-world operating systems, such as demand paging, security features, and more advanced TLB

management strategies. In conclusion, this project helped us grasp the fundamental principles of virtual memory management and the advantages of using a TLB for efficient address translation. It underscores the significance of TLBs in improving system performance by reducing the need to access the full page table. This knowledge can be valuable for those interested in computer architecture, operating systems, and memory management. While this project provided a foundation for understanding virtual memory and TLBs, it also opens the door to more advanced and intricate aspects of memory management in real-world systems. Further exploration could involve implementing more sophisticated TLB management policies and dealing with challenging scenarios, such as page faults and security considerations.

CHAPTER 5

FUTURE SCOPE

The project involving the implementation of virtual memory management with a TLB and page table provides a solid foundation for understanding the fundamentals of memory management in computer systems. There are several avenues for further development and exploration:

- Advanced TLB Management:** Implement more sophisticated TLB management policies such as LRU (Least Recently Used), LFU (Least Frequently Used), or a combination of strategies. Experiment with different replacement algorithms to optimize TLB performance further.
- Demand Paging:** Extend the project to support demand paging, a mechanism where not all pages need to be loaded into physical memory at once. Implement page swapping and paging strategies to load pages into memory only when they are needed.
- Security Features:** Explore the integration of security features, such as memory protection, process isolation, and access control. Enhance the project to address security concerns within a virtual memory system.

- Multi-Level Page Tables:** Implement multi-level page tables to manage address spaces with a larger number of pages efficiently. This can be particularly useful when working with systems that have extensive address spaces.
- Real-World Operating System Features:** Consider incorporating real-world features found in modern operating systems, like shared memory, file mapping, and copy-on-write, to gain a deeper understanding of memory management complexities.
- Performance Optimization:** Optimize the program's performance by considering

system-specific factors, such as cache hierarchies and TLB configurations of different computer architectures. Profiling and tuning your program for specific hardware can be an interesting exploration. Kernel Development: Move beyond a user-level program and venture into kernel development. Develop a simplified operating system kernel that includes memory management features, such as process scheduling, context switching, and system calls.

Parallel and Distributed Systems: Explore memory management challenges in parallel and distributed systems, where multiple processors or machines cooperate in managing memory and addressing data locality issues. Research and Optimization: Consider this project as a starting point for research and optimization in the field of memory management. Investigate new techniques and algorithms that can improve memory management in modern computing environments. Education and Training: Utilize the project as a teaching tool for educational purposes. Create tutorials, documentation, or workshops to help others learn about virtual memory management and memory hierarchies. The future scope of this project is vast, and it can be tailored to align with specific interests and learning goals.

CHAPTER 6

REFERENCES

1. Textbook:

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne, “Operating System Concepts”
10th Edition

2. Websites:

<https://www.geeksforgeeks.org/logical-and-physical-address-in-operating-system/>

<https://workat.tech/core-cs/tutorial/logical-and-physical-address-os-8abv46w3k0bu>

<https://www.geeksforgeeks.org/virtual-memory-in-operating-system/>

<https://www.baeldung.com/cs/virtual-memory-why>

<https://www.spiceworks.com/tech/devops/articles/what-is-virtual-memory/>

APPENDIX

This section contains details on the language, software and packages used in our project. The project is written in the C programming language and primarily uses standard C libraries for file I/O, memory allocation, and basic operations. Let's take a brief look at the language and packages used in this project:

Programming Language: C

C Language: C is a general-purpose, procedural programming language known for its efficiency, low-level memory access, and portability. It is commonly used in systems programming, including operating systems, device drivers, and embedded systems.

Standard C Libraries:

- `stdio.h`: This header file is part of the C standard library and provides functions for input and output operations. It is used for reading addresses from a file, printing messages, and displaying results.
- `stdlib.h`: This header file contains functions for memory allocation and basic operations. It is used for dynamic memory allocation, type conversions, and program termination.

Web-based Linux Simulation:

JSLinux is an online platform that emulates the operation of a complete Linux distribution within a web browser. Users can experience a Linux environment directly within their web browser, without the need for a traditional virtual machine or system installation. JSLinux provides an interactive shell interface where users can execute Linux commands and explore the Linux environment directly in the browser.

Output Module

```
#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;

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

    if(argc<2)
    {
        printf("NOT ENOUGH AGRUMENTS\n EXITING\n");

        return 0;
    }

    fd=fopen(argv[1],"r");

    if(fd==NULL)
    {
        printf("ERROR OPENING FILE\n FILE FAILED TO OPEN\n");

        return 0;
    }

    char *value=NULL;

    size_t len=0;

    ssize_t read;
```

```

long long page_no,offset,page_table,totalhits=0,fault=0,pages=0;

int qp=0;

int physicalad=0,frame,logicalad;

int tlb[TLB_SIZE][2];

int pagetable[PAGE_SIZE];

memset(tlb,-1,TLB_SIZE*2*sizeof(tlb[0][0]));

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

int mask=255,mask1=62580,i,hit;

while((read=getline(&value,&len,fd))!=-1)
{
    pages++;

    //get page number and offset from logical address
    page_no=atoi(value);

    page_no=page_no>>8;

    page_no=page_no & mask;

    offset=atoi(value);

    offset=offset & mask;

    logicalad=atoi(value);

    //printf("%lld %lld\n",page_no,offset);

    frame=0,physicalad=0;

    hit=0;                //1 if found in TLB

    //CHECK IN TLB
    for(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==1)

    printf("TLB HIT\n");


//search in pagetable
else
{

    int f=0;

    for(i=0;i<PAGE_SIZE;i++)

    {

        if(pagetable[i]==page_no)

        {

            frame=i;

            fault++;

            break;

        }

        if(pagetable[i]==-1)

        {

            f=1;

```

```

                                break;
                                }
                                }
                                if(f==1)
                                {
                                        pagetable[i]=page_no;
                                        frame=i;
                                }
                                //replace in tlb using fifo
                                tlb[qp][0]=page_no;
                                tlb[qp][1]=i;
                                qp++;
                                qp=qp%15;
                                }
                                if(logicalad<10000)
                                printf("VIRTUAL ADDRESS = %d \t\t",logicalad);
                                else
                                printf("VIRTUAL ADDRESS = %d \t",logicalad);

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

                                double hitrate=(double)totalhits/pages*100;
                                double faultrate=(double)fault/pages*100;
                                printf("\nTLB HIT RATE= %.2f %c", hitrate,'%');
                                printf("\nTLB MISS RATE= %.2f %c", (100-hitrate),'%');

```

```
printf("\nPAGE TABLE HIT RATE= %.2f %c", fault_rate, '%');  
printf("\nPAGE TABLE MISS RATE= %.2f %c\n", (100-fault_rate), '%');  
}
```