Virtual memory simulator

Student: Violeta Maria Hoza

Group: 30434

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Contents

Introduction

1.1 Context

The aim of this project is to illustrate the key aspects of virtual memory, which is a core concept that allows the abstraction of physical memory[1], allowing the system to simulate more memory than physically available. Virtual memory is a common technique used in a computer's operating system (OS) to manage and optimize the use of limited physical memory (RAM), providing each process with the illusion of a large, continuous memory space. This abstraction is achieved by using virtual addresses, which are mapped to physical addresses by the hardware and OS.

By using virtual memory, systems can:

- compensate for physical memory shortages
- isolate processes to enhance security and process isolation
- support multitasking and the execution of large programs.

To study and understand virtual memory's behaviour, a simulator can be developed to emulate this functionality, allowing for the exploration of page allocation, translation, and replacement policies. This simulator is useful for operating system students, researchers, and developers seeking to understand virtual memory management.

1.2 Objectives

The project's core objective is to simulate the paging process, illustrating the mechanisms involved in virtual memory management such as page tables, page replacement algorithms, and address translation. The simulator will:

- simulate the process of locating information (finding a page): The simulator will handle memory access requests from the user, checking if the requested page is present in main memory (RAM) or needs to be fetched from secondary storage (disk). This will demonstrate the basic concept of page hits and page faults.
- simulate address translation using page tables and TLB (Translation Lookaside Buffer): The simulator will use a Memory Management Unit (MMU) to translate virtual addresses into physical addresses. This process will involve using page tables and a TLB to optimize the speed of address translation and demonstrate how address translation works in systems with virtual memory.

- simulate loading a page into main memory: When a page fault occurs (the page is not in main memory), the simulator will emulate the process of loading the requested page from secondary storage (disk) into the main memory. This involves adding the page to the physical memory while considering memory constraints.
- the simulator will demonstrate the mechanisms involved when replacing a page in memory, using various page replacement algorithms such as FIFO (First-In-First-Out), LRU (Least Recently Used), or others
- provide a user interface for visualizing key operations like memory accesses, page faults, page hits, and the process of loading and replacing pages in the main memory.

The users can set the size of the physical memory, the page size, the number of bits used for the virtual address, and select a page replacement strategy. A graphical view will display the state of the page table, showing which pages are in memory, which have been swapped, and memory accesses. The system will provide feedback on memory access performance, including the number of page hits, page faults, and the efficiency of different page replacement strategies.

Bibliographic Research

2.1 What is Virtual Memory?

Virtual memory is a memory management technique used by operating systems to give the appearance of a large, continuous block of memory to applications, even if the physical memory (RAM) is limited. It allows the system to compensate for physical memory shortages, enabling larger applications to run on systems with less RAM. The main memory can act as a "cache" for the secondary storage, usually implemented with magnetic disks [2]. Virtual memory also allows processes to share files easily and to implement shared memory. In addition, it provides an efficient mechanism for process creation[4].

This concept was first introduced in the Atlas system developed at the University of Manchester in 1959[3] and has since become a cornerstone in modern operating systems such as UNIX, Linux and Windows.

There are 3 main problems that can be solved by using this technique:

- memory shortage: prevents crashes by using paging or swapping mechanisms;

- memory fragmentation: ensures efficient allocation and avoids running out of memory due to fragmentation;

- security: isolates processes and prevents one process from corrupting another's data.

2.2 How does virtual memory work?

Virtual memory uses both the computer's software (the OS) and hardware (typically the Memory Management Unit, or MMU) to work. It transfers processes between the computer's RAM and hard disk by copying any files from the computer's RAM that aren't currently in use and moving them to the hard disk. By moving unused files to the hard disk, a computer frees up space in its RAM to perform current tasks, such as opening a new application. If, at any point, the RAM space is needed for something more urgent, data can be swapped out of RAM and into virtual memory. However, if too many pages are being swapped between the main memory and the hard disk, it can lead to **thrashing** – a state where the computer spends more time swapping pages than executing instructions.

2.3 Types of Virtual Memory

In a computer, virtual memory is managed by the Memory Management Unit (MMU), which is often built into the CPU. The CPU generates virtual addresses that the MMU translates into physical addresses.

There are 2 ways computers handle virtual memory, through paging and segmentation. Sometimes, both paging and segmentation are used together. In this case, memory is divided into pages, and segments are made up of multiple pages. The virtual address includes both a segment number and a page number.

- *Paging**:***

Paging divides memory into small fixed-size blocks called pages. When the computer runs out of RAM, pages that aren't currently in use are moved to the hard drive, into an area called a swap file. The swap file acts as an extension of RAM. When a page is needed again, it is swapped back into RAM, a process known as page swapping. This ensures that the operating system (OS) and applications have enough memory to run.

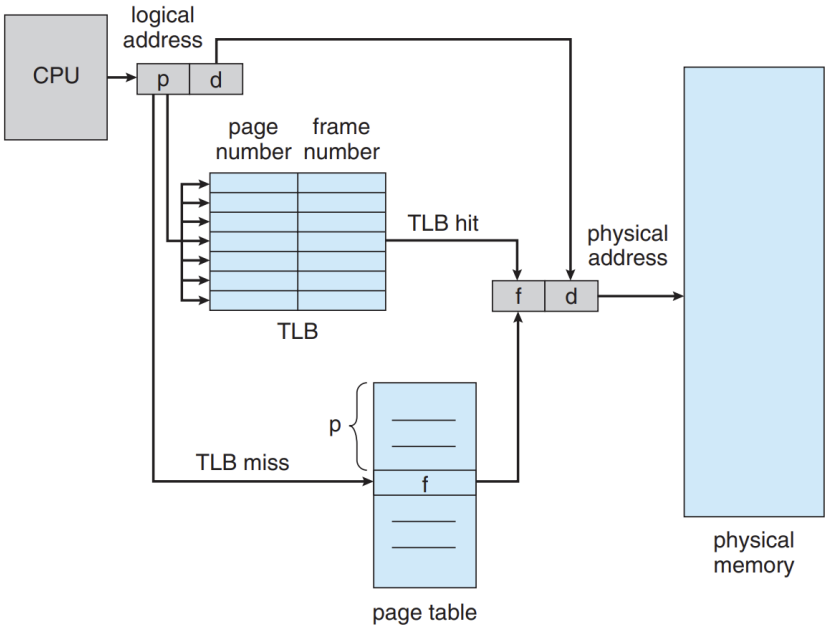


Figure 2.1: Paging hardware with TLB [4]

- *Segmentation:*

Segmentation divides virtual memory into segments of different sizes. Segments that aren't currently needed can be moved from the computer's virtual memory space to its hard drive. The system uses a segment table (a generalization of the base and limit registers used for (single area) contiguous allocation) to keep track of each segment's status, including whether it's in memory, if it's been modified, and its physical address. Each segment is allocated an entry in the segment table; the entry contains the base and limit values for that segment. Segments are mapped into a process's address space only when needed. Segmentation differs from paging because it divides memory into sections of varying lengths, while paging divides memory into units of equal size. With paging, the hardware determines the size of a section, but the user can select the length of a segment in a segmentation system. Segmentation is often slower than paging, but it offers the user more control over how to divide memory and may make it easier to share data between processes.

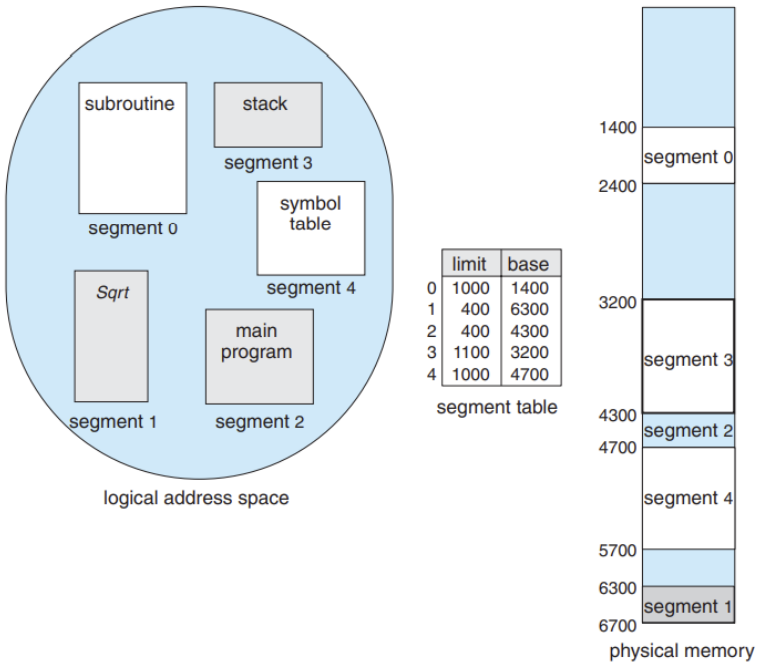


Figure 2.2: Example of segmentation [4]

2.4 Address Translation (Binding)

Address translation involves mapping logical (virtual) addresses generated by processes to physical addresses in RAM. This is achieved through page tables, which store mappings between pages in virtual memory and frames in physical memory. Each process has its own page table, and the operating system maintains the structures needed for these translations. To speed up translation, a small hardware cache [1] known as the **TLB** (Translation Lookaside Buffer) is used to store frequently accessed page table entries. On a TLB miss, the page table is accessed, and if the page is not in memory, a page fault occurs, triggering the operating system to bring the page into RAM from disk.

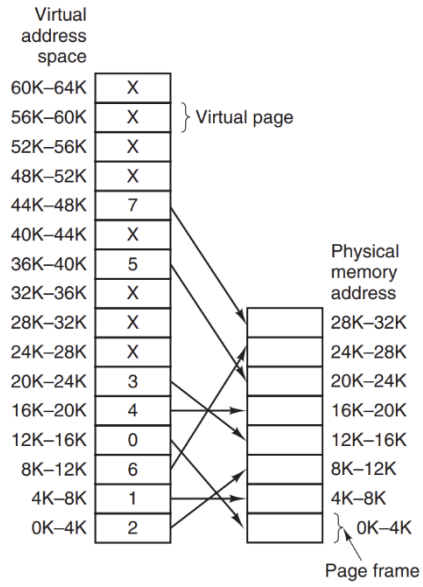


Figure 2.3: The relation between virtual addresses and physical memory addresses is given by the page table. In this example every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287 [5].

2.5 Page Replacement Algorithms

When the physical memory is full or a page fault occurs, a page replacement strategy is needed to determine which page to evict to make room for the incoming page. Several algorithms are used for page replacement:

- **The Optimal Page Replacement Algorithm**: "At the moment that a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later. Each page can be labelled with the number of instructions that will be executed before that page is first referenced. The optimal page replacement algorithm says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible."[5] This algorithm is ideal but impractical as it requires knowledge of future references.
- **FIFO (First-In, First-Out) Page Replacement Algorithm**: Pages are removed in the order they were loaded into memory. A big problem with this page replacement algorithm is the appearance of Belady's anomaly: for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases[4].
- **The Not Recently Used (NRU) Page Replacement Algorithm**: "In order to allow the operating system to collect useful page usage statistics, most computers with virtual memory have two status bits, R and M, associated with each page. R is set whenever the page is referenced (read or

written). M is set when the page is written to (i.e., modified). The bits are contained in each page table entry. [...] The NRU algorithm removes a page at random from the lowest-numbered nonempty class. Implicit in this algorithm is the idea that it is better to remove a modified page that has not been referenced in at least one clock tick (typically about 20 msec) than a clean page that is in heavy use." [5]

- **The Least Recently Used (LRU) Page Replacement Algorithm**: Evicts the page that has not been used for the longest time, assuming that pages used recently will likely be used again soon.
- **The Second Chance Page Replacement Algorithm**: "A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the R bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues." [5]

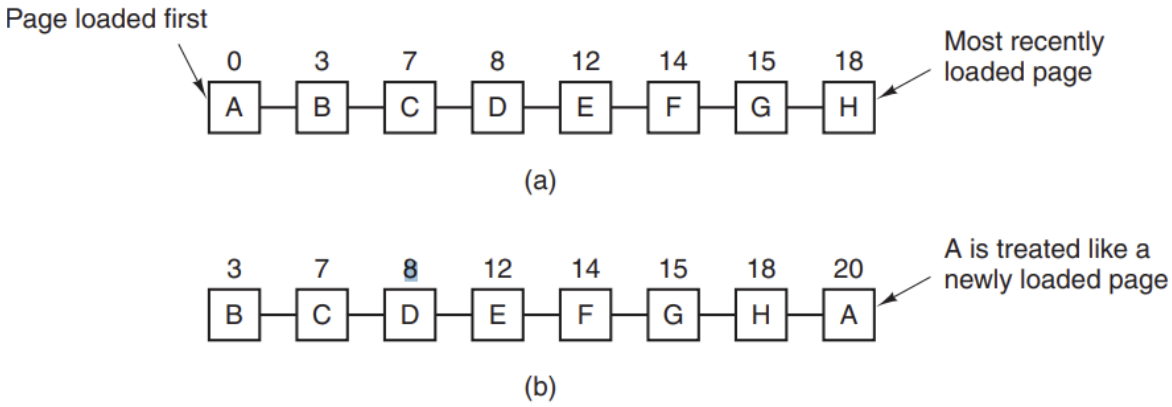


Figure 2.4: Operation of second chance. (a) Pages sorted in FIFO order. [5]

- **The Clock Page Replacement Algorithm**: All the page frames are kept on a circular list in the form of a clock.

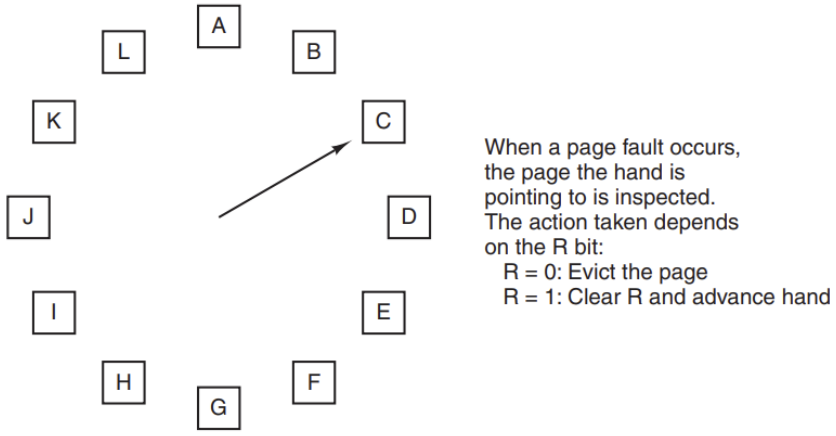


Figure 2.5: The clock page replacement algorithm. [5]

Analysis

3.1 Scenarios

- System parameters configuration: after launching the application, the user sets the virtual address width, page size, physical memory size, TLB size, and selects a page replacement algorithm
- Memory access request: the user inputs a virtual address, and the simulator checks the TLB and page table to determine if the page is in memory and retrieves it, updating statistics on page hits and page faults
- View results: after processing the memory access, the user will see a graphical representation of the memory, page tables, TLB, and current memory access status.

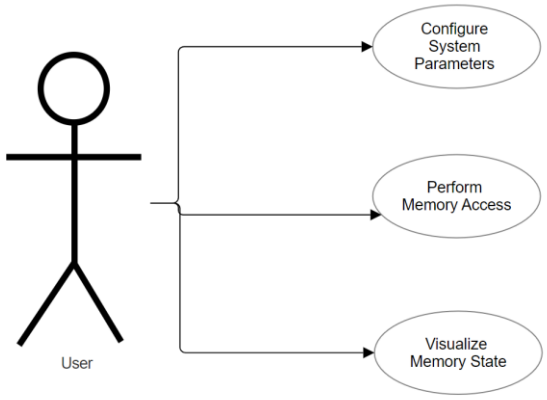


Figure 3.1: Use cases

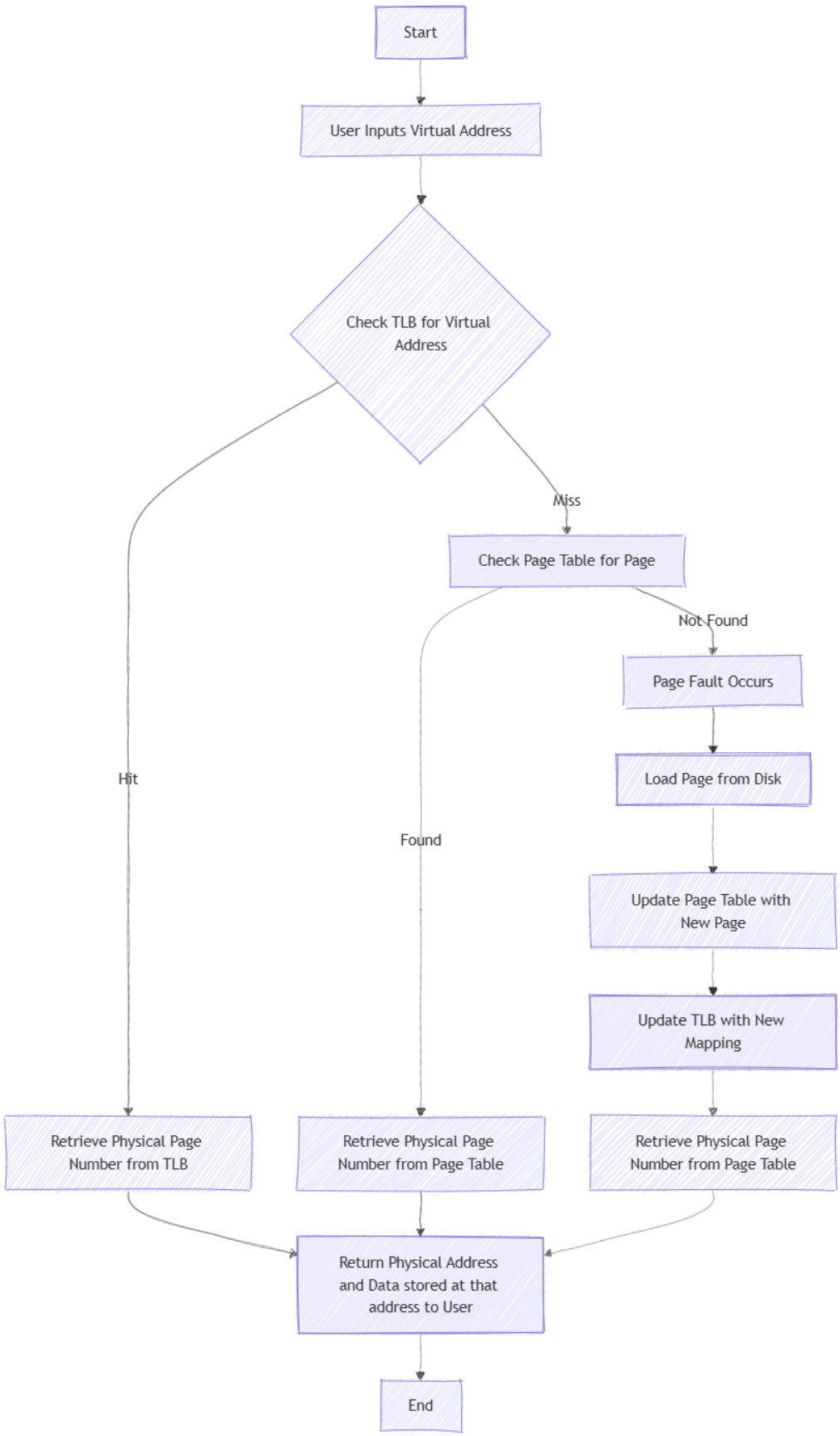


Figure 3.1: The steps followed by the virtual memory simulator when a user tries to access data from a memory address.

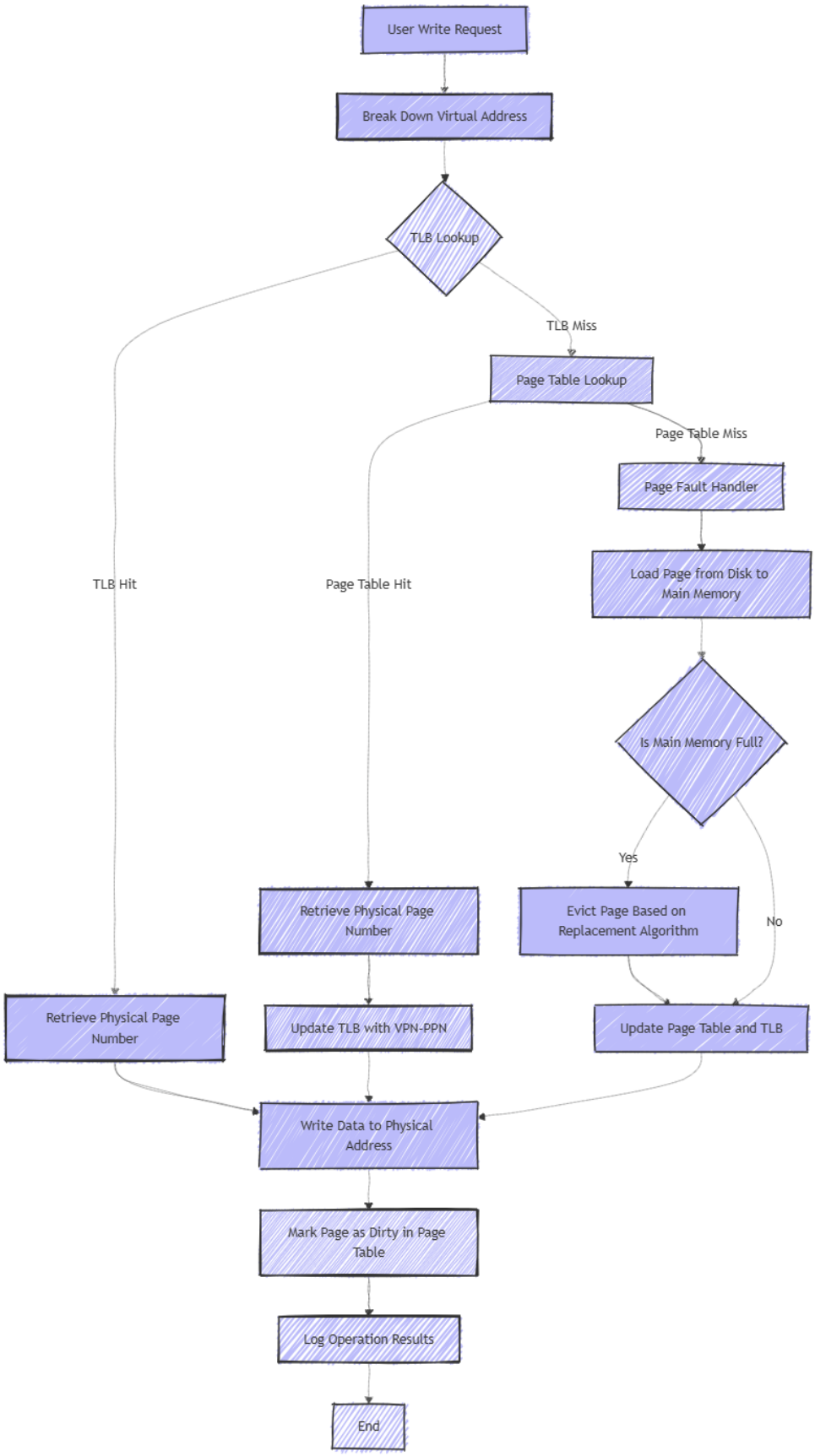


Figure 2.3: The steps followed by the virtual memory simulator when a user writes data to a memory address.

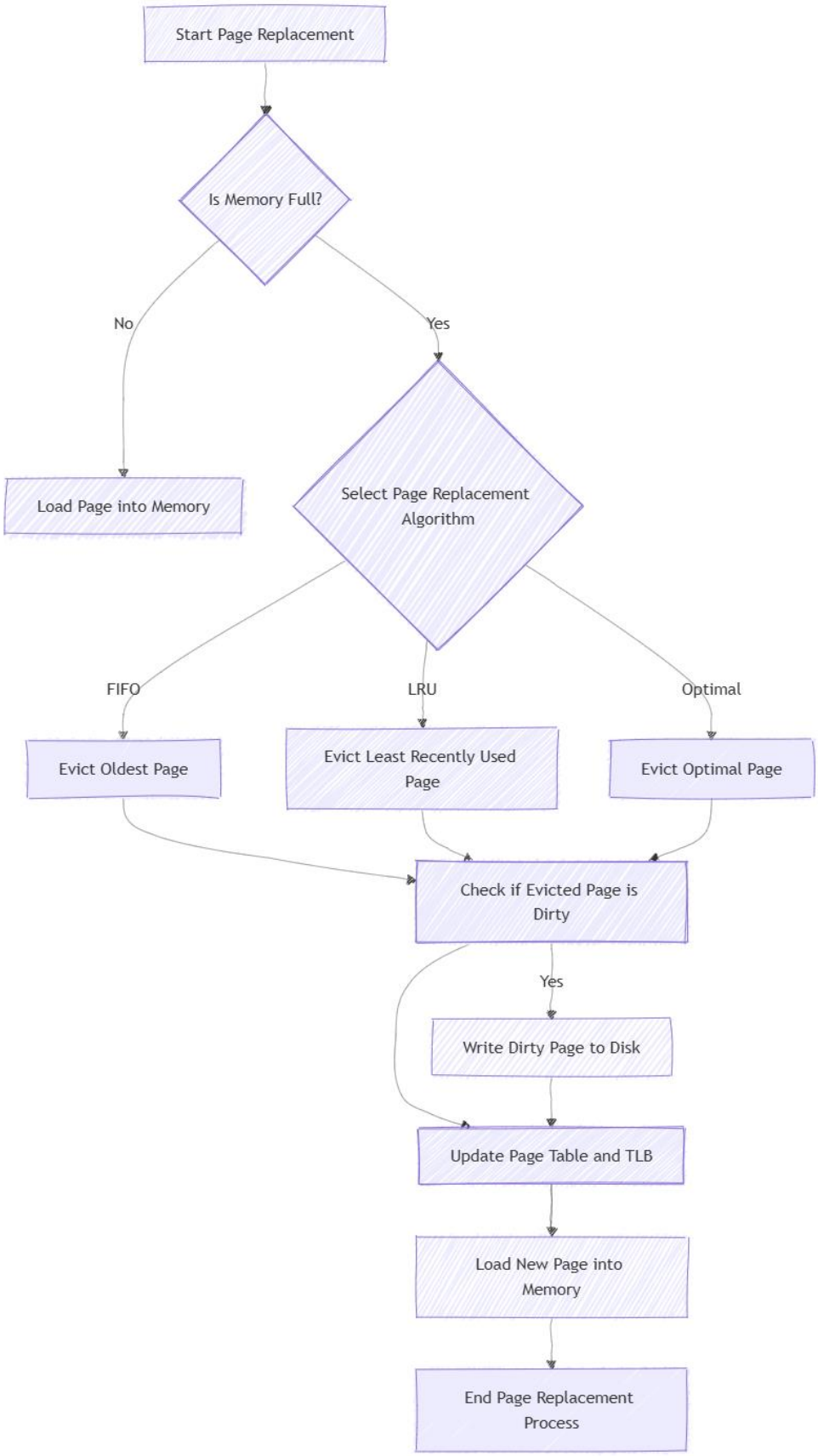


Figure 3.3: Page Replacement Process

3.2 User interaction

After launching the application, the user is met with a welcome screen that displays information about the simulator's purpose and a "Start" button to begin. After clicking "Start," the user is directed to a configuration screen where they can set up the simulation parameters: virtual address width, page size, physical memory size, TLB size, page replacement algorithm. Once all parameters are set, the user clicks a "Configure" or "Start Simulation" button. The user inputs a virtual address they want to access via a designated input field. The user can end the simulation by clicking an "End Simulation" or "Exit" button.

3.3 Expected results

Upon interaction with the simulator, the following results are anticipated:

- Configuration Validation: The simulator will confirm the validity of the configuration settings before proceeding.
- The simulator should track page table / TLB hit and miss counts, count each disk read and write operation and track the number of page evictions.
- Graphical representation or tabular display showing: TLB contents, which pages are currently in memory.
- Operation Logging: The simulator should log each memory operation (load/store) along with its outcome. Logs should specify whether the operation resulted in a TLB hit/miss, page table hit/miss, and if any actions were taken (e.g., page fault, page loaded from disk).

Design

The virtual memory simulator is designed to replicate the behaviour of a computer system's memory management process. It simulates the interaction between key components such as the Page Table, Translation Lookaside Buffer (TLB), Main Memory, Secondary Memory (Disk), and the Memory Management Unit (MMU). The simulator mimics how virtual memory is handled in systems with a limited amount of physical memory and the necessity to swap data in and out of secondary storage when needed.

The design of the virtual memory simulator is structured around several key components that interact to simulate virtual memory management effectively. The simulator consists of the following main components:

- Page table: A data structure that maps virtual addresses to physical frames. Each process has its own page table, which the Memory

Management Unit (MMU) utilizes to translate virtual addresses into physical addresses. The page table is implemented as a hash map, where the key is the virtual page number, and the value is a PageTableEntry object that stores essential data, including frame numbers and valid/invalid bits, dirty and referenced bits. Functionalities: add new page entries when a page is loaded into memory, retrieve a page entry based on a virtual address, indicating whether it is currently in physical memory.

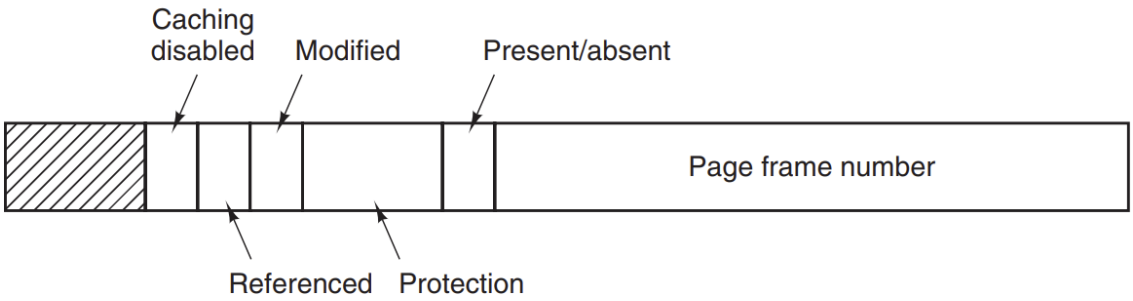


Figure 4.1: A typical page table entry [5]

- TLB: A cache for frequently accessed page table entries. It speeds up the address translation process by allowing the MMU to quickly determine whether a virtual address is mapped to a physical frame without consulting the full page table. A LinkedHashMap is used to store the entries, allowing for efficient access and order maintenance for entry eviction. Functionalities: check if a page is in the TLB when a virtual address is accessed.

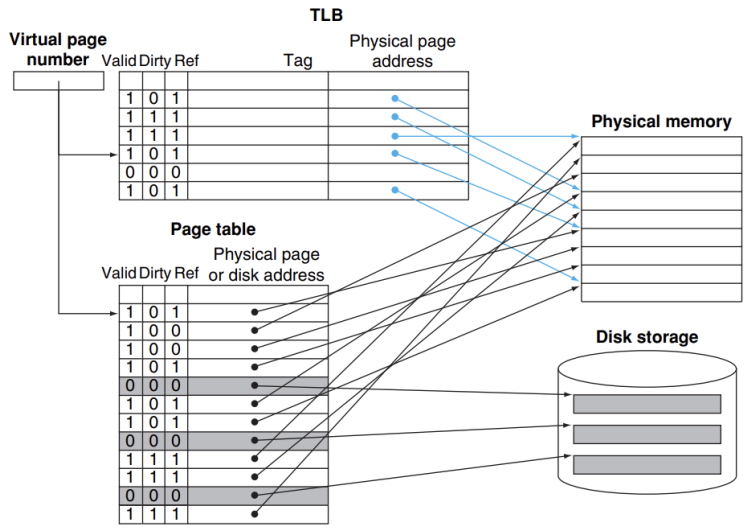


Figure 4.2: The TLB acts as a cache of the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. [2]

- Main memory: Represents physical memory (RAM) where pages are stored. The simulator models the limited physical memory available for storing pages, handling memory access requests accordingly. The main memory is represented as a hash map, where the key is the frame number, and the value is a Page object. Functionalities: load/store pages.
- Secondary memory (disk): Represents secondary storage where pages are fetched if not present in memory. The simulator models a disk where pages are stored when they are swapped out of RAM due to memory constraints. The disk is represented as a map of pages, providing methods to load and store pages.
- Memory Management Unit (MMU): Responsible for translating virtual addresses to physical addresses (the virtual address is divided into a VPN – Virtual Page Number and an offset; the VPN is used to locate the corresponding PPN – Physical Page Number, through the TLB and page table; the physical memory address is obtained by combining the PPN and the offset). Contains methods for checking the TLB and page table, managing page faults, and loading / storing pages into memory.

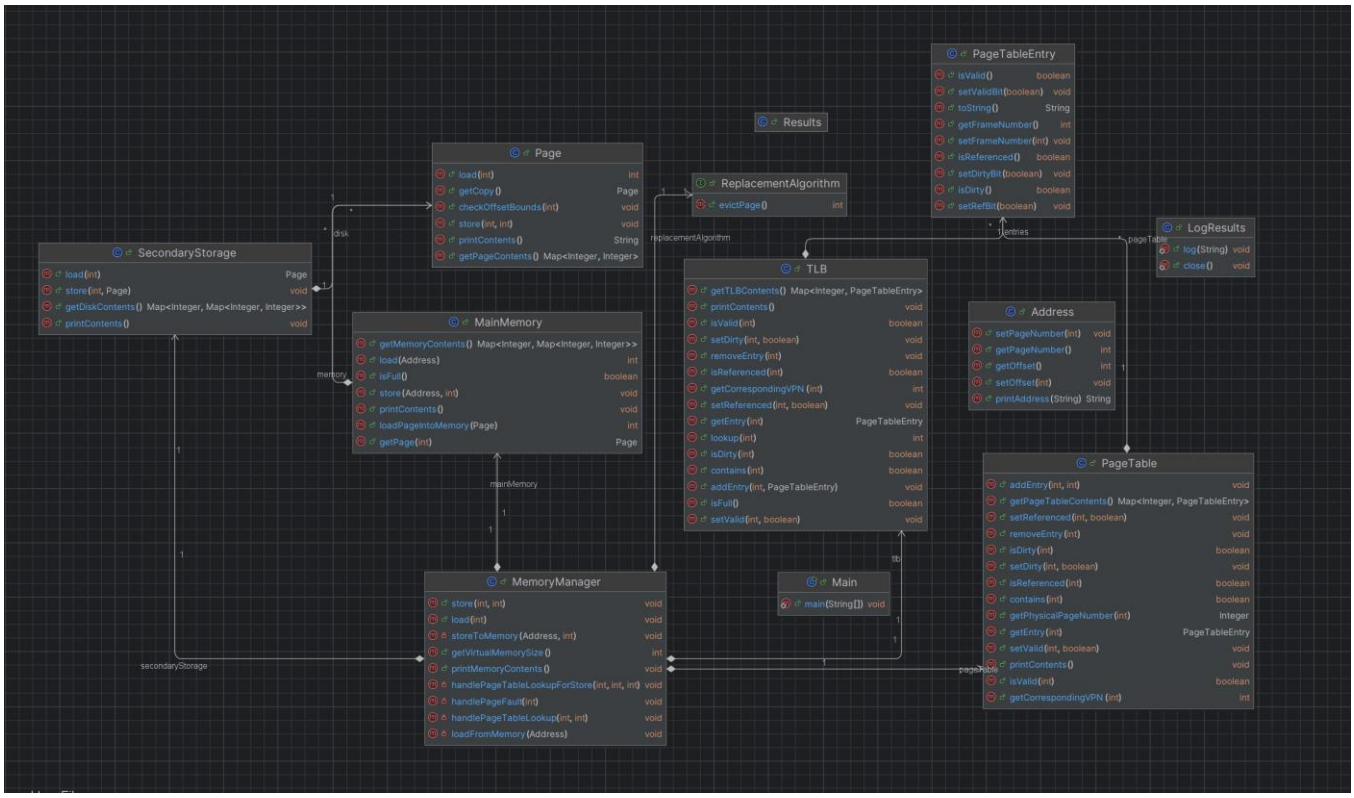


Figure 4.3: Initial design

Implementation

The implementation of the virtual memory simulator in Java involves defining classes for each component outlined in the design phase. Key classes include: Address, Page, PageTableEntry, PageTable, TLB, MainMemory, SecondaryStorage, MemoryManager, ReplacementAlgorithm.

Some libraries and frameworks that can be used to implement the simulator are:

- Junit: for testing different scenarios, such as TLB hits and page faults, ensuring that the memory manager behaves as expected.
- JavaFX or Swing for the GUI

The GUI for the Virtual Memory Simulator leverages several libraries and frameworks to ensure a user-friendly and responsive interface:

- Bootstrap 5 (CSS Framework): used for structuring the layout of the page.
- Font Awesome (Icon Library): icons are used next to buttons for a more visually appealing and user-friendly experience.
- Thymeleaf (Template Engine): dynamic data binding (The th:text attribute in the HTML is used to bind server-side data to the page. For example, the TLB hit rate, page table miss rate, and other simulation results are populated dynamically from the server-side data model.), form binding (The th:value attribute is used to bind input fields with server-side values, allowing the form fields to retain user inputs between requests or when the page reloads after form submission.)
- JavaScript (Client-Side functionality): JavaScript is used to dynamically calculate and display the virtual memory size and page table size when the user changes the virtual address width or page size. This is done using the updateMemoryDetails() function, which listens for input events on the corresponding fields. JavaScript also handles form validation before submission. The validateInputs() function ensures that the physical memory size, page size, and disk size are powers of 2, and that other constraints (like disk size being larger than physical memory) are met. If any condition is violated, an alert message is shown.
- CSS: for custom styling.

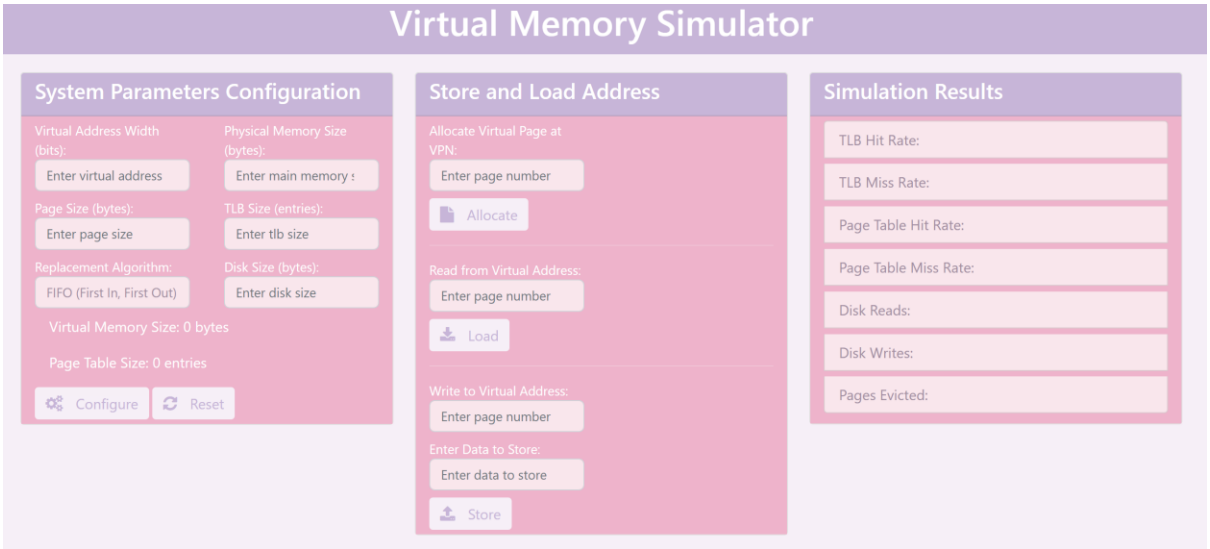


Figure 5.1: First view of the GUI

5.1 Model Package

The model package in the Virtual Memory Simulator application contains the data structures and logic that represent and manage the state of the virtual memory system. This package is crucial as it encapsulates the core functional components of the simulator, providing a foundation upon which the simulation operates. Here is a detailed look at each class within this package:

5.1.1 Address

The Address class represents an address in memory, which can either be a virtual address or a physical address. The class encapsulates two important components of an address: the page/frame number and the offset within the page/frame.

Key functionalities:

- Page Number and Offset: The address is divided into two parts - the page number (VPN or PPN) and the offset (a position within a page or frame). The page number is used to identify a specific page or frame in memory, while the offset identifies the exact location within that page/frame.
- Constructors: The constructor initializes the pageNumber and offset properties with the values passed as arguments. This allows for the creation of either virtual or physical memory addresses based on the context.
- Getter and Setter methods:
    - The getPageNumber() method returns the page/frame number of the address.

    - The getOffset() method returns the offset within the page/frame.
    - The setPageNumber(int pageNumber) method sets the page/frame number.
    - The setOffset(int offset) method sets the offset within the page/frame.
- String representation: The printAddress() method provides a string representation of the address based on its type, either as a virtual address or a physical address. This is useful for debugging and logging the access requests for virtual or physical memory locations.

5.1.2 Page

The Page class represents a page in virtual memory, which stores data in offset-data pairs. Each page contains a collection of offsets, where each offset corresponds to a specific piece of data. The class is designed to manage the contents of a page and provides methods for accessing, storing, and manipulating the data contained within the page.

Key functionalities:

- Fields:
  - The pageContents field is a LinkedHashMap that stores data as offset-data pairs, where the key is the offset within the page and the value is the data at that offset. This structure allows efficient storage and retrieval of data from specific offsets within the page.
  - The pageSize field represents the number of offsets that a page can hold. The page size is initialized when the page is created and dictates how many pieces of data can be stored in that page.
- Constructors:
  - Public Constructor: The constructor initializes a page with a specified size, creating the pageContents map and filling it with default data where the offset corresponds to its value.
  - Private Constructor: A private constructor is provided for creating a copy of an existing page. It performs a deep copy of the page contents to ensure isolation between the original and the cloned page.
- Offset validation: The method checkOffsetBounds(int offset) checks whether a given offset is within the valid range of the

page. If the offset is out of bounds, a log is generated. This ensures that data is not accessed or modified outside the valid range of the page.

- Loading and Storing data:
  - The load(int offset) method retrieves the data stored at a specific offset in the page. If the offset is valid, it returns the value stored at that offset. If the offset is invalid, the method logs an error.
  - The store(int offset, int value) method stores a given value at a specified offset in the page. Like load, it first checks if the offset is valid before proceeding to store the data.
- Page content access: The method getPageContents() returns a copy of the current page's contents. This prevents external code from directly modifying the original data in memory, maintaining the integrity of the page's contents.
- Page copy: The getCopy() method creates and returns a new instance of the Page class with identical contents. This is useful when a duplicate of the page is required, such as when swapping pages or handling multiple processes.
- Printing page contents

5.1.3 PageTableEntry

The PageTableEntry class represents a single entry in the page table/TLB of the virtual memory system. Each entry holds critical information about a page's location in physical memory, its state, and its validity. This class encapsulates the attributes necessary to manage virtual memory effectively, including the frame number, validity, dirty and reference status, and whether the page is stored on disk.

Key functionalities:

- Fields:
  - The frameNumber field stores the physical page number (also known as the physical page number or PPN). If the page is currently not loaded into memory, this is set to -1, indicating no valid mapping.
  - The validBit field is a boolean that indicates whether the page is currently in main memory. If the page is in memory, the valid bit is true; otherwise, it is false.
  - The dirtyBit field is a boolean that tracks whether the page has been modified while in memory. If the page is modified after being loaded into memory, the dirty bit is set to true,

indicating that the page needs to be written back to disk if it is swapped out.
- The refBit field is a boolean that is set whenever the page is referenced, either for reading or writing. This is essential for memory management algorithms like the Clock or LRU (Least Recently Used) algorithms, which use the reference bit to determine which page to swap out when memory is full.
- The diskPage field is a boolean that indicates whether the page is currently stored on disk (secondary storage) due to being swapped out of memory. If true, the page is on disk; if false, it is in memory.
- accessTime: A long field representing the last time the page was accessed. This is used by LRU and other time-based replacement algorithms to determine which page to evict when memory is full.

- Constructors: default constructor, parameterized constructor
- Getter and setter methods
- toString() method

5.1.4 PageTable

The PageTable class represents a page table used in virtual memory management. It maintains a mapping between virtual page numbers (VPN) and physical page numbers (PPN). It manages page table entries (i.e., instances of PageTableEntry), each of which holds various attributes regarding a page's state (e.g., whether it is in memory, whether it has been modified, etc.). This class provides methods to manage, retrieve, and update page table entries, and supports logging of key events related to the page table's operation.

Key functionalities:

- PageTable structure: The class stores a pageTable, which is a map (Map<Integer, PageTableEntry>), mapping each virtual page number (VPN) to its corresponding page table entry (PageTableEntry). The size attribute defines the number of entries in the page table.
- Constructor: PageTable(int size): Initializes the page table with a specified number of entries. Each entry is initially created using the default constructor of PageTableEntry, meaning that each entry starts with default values (e.g., invalid, not referenced, not dirty).

- Retrieve entries, add entries
- Page entry state management (isValid(int vpn), setValid(int vpn, boolean valid), etc.)
- Print contents, get page table contents
- Access time management: the updateAccessTime method integrates with the LRU replacement algorithm to synchronize the last access time of each page. This ensures that page table entries reflect the most recent access times.

5.1.5 TLB

This TLB (Translation Lookaside Buffer) class manages the translation of virtual page numbers (VPN) to their corresponding page table entries. It uses a specified eviction algorithm to manage its size, evicting entries when full, and offers several methods to access, modify, and inspect its contents.

Key functionalities:

- LinkedHashMap for Entries: The TLB uses a LinkedHashMap to store mappings of VPNs to PageTableEntry. This allows efficient lookups and ordering when eviction happens.
- Eviction handling: The class supports different eviction algorithms by allowing the use of a ReplacementAlgorithm to manage which page is evicted when the TLB reaches its maximum size.
- Methods for managing entries (getEntry(int vpn), addEntry, removeEntry, containsEntry, etc.)
- Page entry state management

5.1.6 MainMemory

The MainMemory class simulates the loading, storing, and management of pages in a virtual memory system. It provides an abstraction for managing physical memory, where pages are stored in memory frames. This class also supports operations such as loading data from memory, storing data to memory, and managing pages using physical addresses.

Key functionalities:

- The main memory is represented as a Map<Integer, Page>, where each frame number (integer) maps to a Page object. A page contains data of a fixed size (the page size), and multiple

pages are stored in memory frames. The number of frames and the size of each page are configurable at initialization.

- Load and store data:
  - load (Address address): Retrieves the data stored at a given physical address by first finding the appropriate page and then accessing the data at the offset.
  - store (Address address, int data): Stores the specified data at the given physical address by finding the corresponding page and offset.
- Load and remove pages:
  - loadPageIntoMemory(Page page, int frameNr): Loads a page into a specific frame number in memory.
  - removePage(int frameNumber): Removes a page from the specified frame.
- Memory status: isFull(), getNextAvailableFrame()
- Page retrieval

5.1.7 SecondaryStorage

The SecondaryStorage class simulates a secondary storage (like a disk) for pages in a virtual memory system. This class provides the ability to store, load, and check for the existence of pages using virtual page numbers (VPNs). It also offers functionality to retrieve the entire contents of secondary storage, which includes mapping virtual addresses to data.

Key functionalities:

- Storage representation: The storage is represented as a Map<Integer, Page>, where the key is the virtual page number (VPN), and the value is a Page object. This map simulates the disk that stores pages, each of which corresponds to a virtual page in the system.
- Methods for managing pages:
  - store(int vpn, Page page): Stores the provided Page in secondary storage at the specified virtual page number (VPN).
  - load(int vpn): Loads a page from secondary storage using its VPN. If the page is found, it is returned; otherwise, an error message is logged, and null is returned.
  - containsPage(int vpn): Checks if a page with the specified VPN exists in secondary storage. Logs the result (whether the page is found or not).

- getDiskContents(): Retrieves the entire contents of secondary storage. This method converts each page's data into virtual addresses, which are calculated as VPN * pageSize + offset. The result is a map of virtual addresses and their associated data.
- printContents()

5.1.8 MemoryManager

The MemoryManager class simulates the management of virtual memory in a computer system. It coordinates the interaction between several components of a virtual memory system, including the Translation Lookaside Buffer (TLB), page table, main memory, secondary storage (disk), and page replacement algorithms.

Key functionalities:

- Constructor: Initializes various components (TLB, PageTable, MainMemory, SecondaryStorage) based on the provided memory configuration, including virtual address width, page size, physical memory size, and disk size.
- Page table lookup: When a virtual address is accessed, the memory manager first checks the TLB for a fast lookup. If the page is not found in the TLB (a miss), it checks the page table. If the page is not in the page table, a page fault occurs, and the page is loaded from secondary storage.
- Page replacement: When the main memory is full, the MemoryManager uses the replacement algorithm (like FIFO) to determine which page to evict. If the evicted page is dirty, it is written back to secondary storage.
- Load and store methods:
  - load(int virtualAddress): Loads data from a given virtual address. It first checks the TLB for a translation. If the translation is not found (TLB miss), it checks the page table and handles any page faults by loading the page from secondary storage.
  - store(int virtualAddress, int data): Stores data at a specified virtual address. Similar to load(), it checks the TLB first and then the page table, handling page faults if necessary.
  - loadFromMemory(Address physicalAddress): Loads data from memory at the given physical address, updates the page's reference bit, and tracks page access for the replacement algorithm.

- storeToMemory(Address physicalAddress, int data): Stores data in memory at the given physical address and marks the page as dirty (if modified) and referenced.

- allocatePage(int vpn): Allocates a page for a specified virtual page number (VPN). It attempts to map it to a physical frame in memory. If there are no available frames, the page is stored in secondary storage.

5.1.9 ReplacementAlgorithm

The ReplacementAlgorithm interface defines the essential methods that any page replacement algorithm (e.g., FIFO, LRU) must implement in a virtual memory management system.

Methods:

- evictPage(): This method is responsible for determining which page should be evicted from memory when there is a need to free up space (e.g., when main memory is full). The eviction strategy varies depending on the specific replacement algorithm being used (e.g., First-In-First-Out (FIFO), Least Recently Used (LRU)).
- addPage(int vpn): This method adds a new page (represented by its virtual page number vpn) to the algorithm's tracking system. It keeps track of which pages are in memory so that eviction decisions can be made later.
- updatePageAccess(int vpn): This method updates the state of a page when it is accessed. For algorithms like LRU (Least Recently Used), this is important to track when pages are last accessed, so the algorithm can decide which pages to evict based on their recent usage.

5.1.10 FIFOReplacement

The FIFOReplacement class is an implementation of the ReplacementAlgorithm interface that follows the First-In-First-Out (FIFO) page replacement policy. The class simulates a FIFO-based page replacement mechanism where pages are evicted in the order they were added. The first page to enter the memory is the first page to be evicted when memory is full.

Key components:

- pageQueue: A Queue<Integer> (specifically a LinkedList) is used to hold the virtual page numbers (VPNs) in the order they were

added. This allows for an efficient FIFO structure where the oldest page is always at the front of the queue.

- maxSize: The maxSize variable defines the maximum number of pages that can be held in memory at once (i.e., the size of the FIFO queue). If the number of pages exceeds maxSize, the FIFO policy evicts the oldest page to make space for the new page.

Methods:

- evictPage(): Evicts the page that has been in memory the longest (the one at the front of the queue). If the queue is empty, the method logs and returns -1, indicating no page can be evicted. If there are pages in the queue, the oldest page (the one at the front) is removed using poll() and returned.
- addPage(int vpn): Adds a new page to the FIFO queue. If the page (vpn) is already in the queue, it does nothing. If the queue is full (i.e., it contains maxSize pages), the method evicts the oldest page using evictPage() before adding the new page. The new page is added to the end of the queue using offer(), which is an efficient way to enqueue elements in a LinkedList.
- updatePageAccess(int vpn): FIFO does not need to track page accesses because the eviction policy does not depend on recency of use, only on the order of arrival. Therefore, this method is a no-op (does nothing).

5.1.11 LRUReplacement

The LRUReplacement class implements the Least Recently Used (LRU) page replacement algorithm, which is a more efficient method of managing memory compared to FIFO, especially when the access pattern is not uniformly distributed.

LRU Policy: The idea behind LRU is that when memory is full, the page that hasn't been used for the longest period (i.e., the least recently used) should be evicted to make room for a new page.

Key components:

- pageAccessTime: A LinkedHashMap<Integer, Long> is used to keep track of the pages and their corresponding access times (represented by a timestamp or counter). The key is the VPN (virtual page number), and the value is the timestamp of the last access. The LinkedHashMap is ordered based on access order, which allows for easy traversal when searching for the least recently used page.

- accessCounter: This counter is incremented each time a page is accessed. It serves as a unique timestamp for each access event, where a higher value represents a more recent access.

Methods:

- evictPage(): Evicts the page that has not been used for the longest time (the LRU page). If the pageAccessTime map is empty, return -1, indicating that no pages are available for eviction. Iterate through the pageAccessTime map to find the page with the oldest access timestamp (the least recently used). Remove the least recently used page from the map and return its VPN.
- addPage(int vpn): Adds a page to memory or updates its access timestamp if it already exists. The updatePageAccess(vpn) method is called, which will update the access time for the given page VPN. This ensures the page is marked as recently used.
- updatePageAccess(int vpn): Updates the access timestamp for a page. The page's VPN is added (or updated) in the pageAccessTime map with a new access timestamp, which is the incremented value of the accessCounter. This update indicates that the page has been accessed recently.

5.1.12 NRUReplacement

The NRUReplacement class implements the Not Recently Used (NRU) page replacement algorithm, which classifies pages based on their referenced and modified bits. The goal of the algorithm is to evict pages based on these classifications, prioritizing the pages that are the least useful according to their access and modification history.

Key concepts:

- Page Classification: Pages are classified into four categories (classes 0-3) based on two bits: the referenced (R) bit and the modified (M) bit.
  - Class 0: Not referenced, Not modified.
  - Class 1: Not referenced, Modified.
  - Class 2: Referenced, Not modified.
  - Class 3: Referenced, Modified.
- Eviction Strategy: The algorithm tries to evict pages from the lowest class first. If no pages can be found in the lowest class, it

proceeds to the next class, and so on. If no pages are found in any class, a random page is evicted.

Key components:

- pageTable: A reference to the PageTable object that holds the page entries (PageTableEntry). Each page has a referenced and modified (dirty) bit.
- random: A Random object is used to randomly select a victim page from a list of eligible pages, in case there are multiple pages available in the same class.
- activePages: A list that keeps track of all active pages (pages that are currently in memory).

Methods:

- evictPage(): It iterates over all four classes (0-3) to find pages that belong to each class. For each class, it checks the pages in the active list and classifies them using the getPageClass method. Once pages of a particular class are found, a random page from that class is selected and evicted. If no pages are found in any class, a random page from the active list is evicted.
- getPageClass(PageTableEntry entry): Determines the class of a given page based on its referenced and modified bits. If the page is referenced (R bit is set), it contributes to classes 2 or 3. If the page is modified (M bit is set), it contributes to classes 1 or 3. Returns an integer between 0 and 3, representing the class of the page.
- addPage(int vpn): Adds a new page to the list of active pages and updates its access status.
- updatePageAccess(int vpn): Updates the page's referenced bit when it is accessed.

5.1.13 OptimalReplacement

The OptimalReplacement class implements the Optimal Page Replacement algorithm, which is a theoretical page replacement strategy that evicts the page that will not be used for the longest period of time in the future. This algorithm is considered the best in terms of minimizing page faults, as it makes the most optimal decision based on future memory access patterns. However, it cannot be implemented in a real system because it requires knowledge of future memory accesses, which is typically unavailable. Despite this, it can

be simulated in a controlled environment or used for academic purposes.

Key components:

- futureAccesses (List<Integer>): A list that stores the future memory access pattern. The entries represent the sequence of Virtual Page Numbers (VPNs) that will be accessed in future steps. The algorithm uses this list to determine which page to evict.
- activePages (Map<Integer, Integer>): A map that tracks the pages currently in memory (active pages). The keys are the VPNs of the pages, and the values are the steps at which these pages were last accessed.
- currentStep: An integer that keeps track of the current simulation step, which is incremented each time a page is accessed.

Methods:

- setFutureReferences: Sets the future memory access patterns for each page. This method must be called before the eviction process to provide the future access information needed for the algorithm.
- evictPage(): Evicts the page that will be used the furthest in the future (or not at all) based on the future access pattern. It iterates over the active pages and checks the next time each page will be accessed. The page with the furthest future use (or no future use) is selected for eviction. If no page is selected, an exception is thrown, which ideally shouldn't happen in a controlled simulation.
- getNextUse(int vpn) : Determines when a page will be accessed next, based on its future access times. If the page has no future accesses, it is considered as having an access time of Integer.MAX_VALUE (i.e., it won't be used again). If the page is going to be accessed again, the method finds the first access time that is later than the current step.
- addPage(int vpn): Adds a page to the active pages list at the current simulation step. This method is called when a page is accessed or loaded into memory.
- updatePageAccess(int vpn): It increments the currentStep to simulate the passage of time. It updates the last access time for the given page (VPN).

5.1.14 Results

This class is responsible for tracking various statistics related to the Translation Lookaside Buffer (TLB) and page table operations, including hit rates, miss rates, and disk access operations. It also includes methods for calculating and logging these statistics, as well as resetting them for new simulations.

5.2 Controller Package

5.2.1 VirtualMemorySimulatorApplication

The VirtualMemorySimulatorApplication class is the entry point for the Spring Boot application that simulates virtual memory management.

5.2.2 MemoryController

The MemoryController class is part of a Spring-based application designed to simulate a virtual memory system, which includes managing memory configurations, simulating memory operations, and resetting the simulation. It serves as the backend controller in the MVC architecture, interacting with the front-end user interface to perform memory management tasks. The controller interacts with a MemoryManager that handles the underlying operations such as page loading, storing, allocation, and eviction. It provides an interface for users to interact with the simulation through a web interface.

5.3 Utils

5.3.1 LogResults

The LogResults class is responsible for logging messages related to the simulation. These logs include details on memory allocation, page mapping, load/store operations, and replacement algorithm activity. Any errors or inconsistencies during simulation execution are captured and logged for debugging and validation purposes.

5.3.2 ConfigLoader

The ConfigLoader class is responsible for loading configuration files in JSON format and retrieving the names of available configuration files from the file system. It provides utility methods to load the configuration file into a SimulationConfig object and to fetch a list of configuration files available for use in the system. This class leverages the Jackson ObjectMapper for deserialization and Spring MVC's @GetMapping annotation to expose an endpoint for fetching the configuration files.

5.3.3 Operation

The Operation class represents a single operation that can be performed within the context of the virtual memory simulation. It encapsulates the details of an operation, which can either be an **Allocate**, **Load**, or **Store** operation. These operations are essential for simulating virtual memory management, where the system interacts with virtual pages and addresses. The class provides getter and setter methods to access and modify the properties of the operation, including the type of operation, address, virtual page number (VPN), offset, and data to be stored.

5.3.4 SimulationConfig

The SimulationConfig class represents the configuration settings used for simulating a virtual memory system. This class contains key parameters such as virtual address width, page size, TLB size, physical memory size, secondary memory size, and the replacement algorithm used for page replacement. Additionally, it stores a list of operations that are part of the simulation, such as Load, Store, and Allocate operations. These configurations are read from a configuration file (e.g., JSON) to control the behaviour of the simulation.

Testing and Validation

In the interface, the user can set the parameters of the simulation: the width of the virtual address, the size of the main memory, the size of the secondary storage, the page size, the TLB size and the replacement algorithm that will be used. The virtual memory size and page table size will be computed automatically according to the input values. Once the user inputs the necessary parameters, the system will initialize the main memory and page table according to the provided sizes. By default, the disk and page table will be populated with default values (e.g., empty or zeroed entries). The updated configuration is stored and used to manage the simulation's behaviour.

Additionally, the simulation supports loading configuration files in JSON format, enabling users to quickly set up predefined configurations without manually entering all values. This functionality is provided through the ConfigLoader class, which reads the configuration from a file and parses it into a SimulationConfig object. The user can load a configuration file containing the desired simulation parameters, making the process more efficient.

After loading the configuration, the user must allocate virtual pages. The allocation process involves mapping the first few virtual pages to the physical pages in the main memory. Any remaining virtual pages will be stored in the secondary storage (e.g., a hard disk or SSD). This ensures that the system operates within the limits of available physical memory, using secondary

storage as an overflow. Once the virtual pages are allocated and mapped, the user can perform load and store operations on the memory. The simulation allows data to be loaded from and stored into both main memory and secondary storage, simulating real-world memory access patterns.

Throughout the simulation, validation is performed at each step to ensure that the memory system behaves as expected:

- **Page Allocation Validation**: After allocating virtual pages, the system verifies that pages are correctly mapped to physical memory, and any excess pages are correctly placed into secondary storage.
- **Load and Store Validation**: During load and store operations, the system checks that the correct data is being read or written to the appropriate memory location, either in physical memory or secondary storage.
- **Configuration Validation**: After loading a configuration file, the system ensures that all parameters (virtual address width, page size, TLB size, etc.) are set correctly, and any derived values (e.g., virtual memory size, page table size) are computed as expected.

The virtual memory simulator was tested to ensure correct functionality and reliability. Key tests include:

1. **Memory Operations**:
    - **Load/Store**: Valid addresses are correctly loaded, and data is stored in memory, as confirmed by the tests for loading valid addresses and storing data.
    - **Invalid Addresses**: The simulator handles invalid address accesses without errors, ensuring stability.
2. **Page Replacement Algorithms**:
    - **FIFO**: Verified that the least recently used page is evicted when memory is full, as tested in the FIFO replacement test.
    - **LRU**: Ensured the least recently used page is evicted under the LRU policy when memory is full.
    - **NRU**: Tested the NRU policy to ensure pages with low usage are evicted correctly.
3. **Edge Cases**: Tests ensure the system handles edge cases, like invalid accesses and eviction scenarios, without failure.

Overall, the simulator was validated across various configurations, confirming that memory operations and page replacement algorithms function as expected.

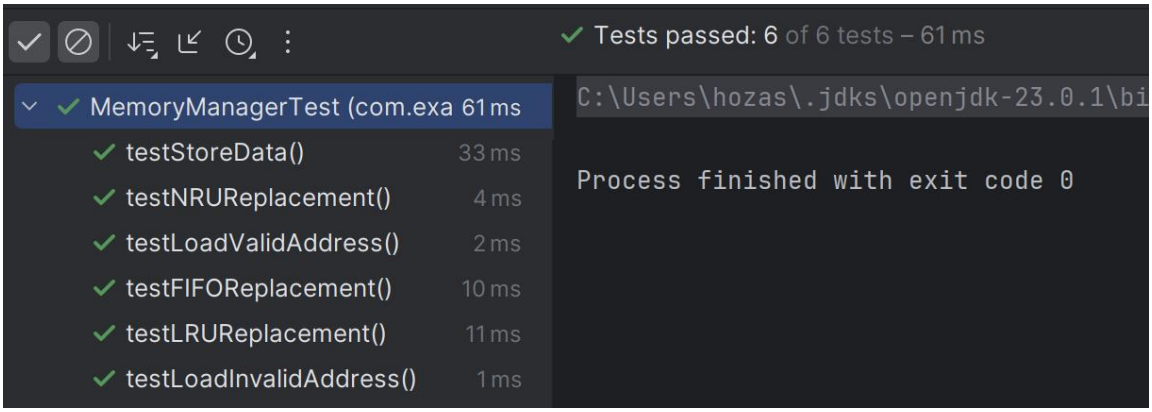
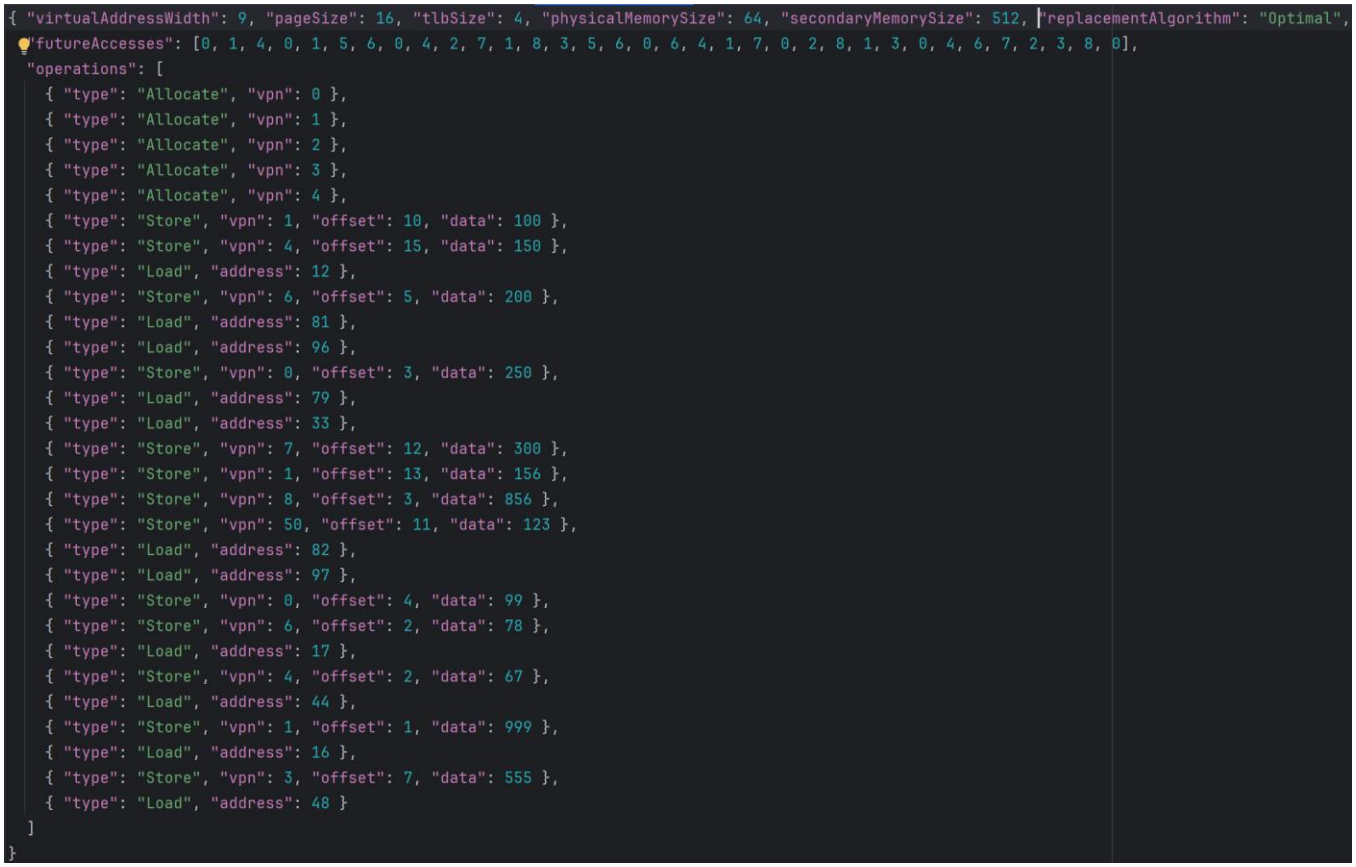


Figure 6.1: Tests results

For the following configuration:



```
{ "virtualAddressWidth": 9, "pageSize": 16, "tlbSize": 4, "physicalMemorySize": 64, "secondaryMemorySize": 512, "replacementAlgorithm": "Optimal",
 "futureAccesses": [0, 1, 4, 0, 1, 5, 6, 0, 4, 2, 7, 1, 8, 3, 5, 6, 0, 6, 4, 1, 7, 0, 2, 8, 1, 3, 0, 4, 6, 7, 2, 3, 8, 0],
 "operations": [
   { "type": "Allocate", "vpn": 0 },
   { "type": "Allocate", "vpn": 1 },
   { "type": "Allocate", "vpn": 2 },
   { "type": "Allocate", "vpn": 3 },
   { "type": "Allocate", "vpn": 4 },
   { "type": "Store", "vpn": 1, "offset": 10, "data": 100 },
   { "type": "Store", "vpn": 4, "offset": 15, "data": 150 },
   { "type": "Load", "address": 12 },
   { "type": "Store", "vpn": 6, "offset": 5, "data": 200 },
   { "type": "Load", "address": 81 },
   { "type": "Load", "address": 96 },
   { "type": "Store", "vpn": 0, "offset": 3, "data": 250 },
   { "type": "Load", "address": 79 },
   { "type": "Load", "address": 33 },
   { "type": "Store", "vpn": 7, "offset": 12, "data": 300 },
   { "type": "Store", "vpn": 1, "offset": 13, "data": 156 },
   { "type": "Store", "vpn": 8, "offset": 3, "data": 856 },
   { "type": "Store", "vpn": 50, "offset": 11, "data": 123 },
   { "type": "Load", "address": 82 },
   { "type": "Load", "address": 97 },
   { "type": "Store", "vpn": 0, "offset": 4, "data": 99 },
   { "type": "Store", "vpn": 6, "offset": 2, "data": 78 },
   { "type": "Load", "address": 17 },
   { "type": "Store", "vpn": 4, "offset": 2, "data": 67 },
   { "type": "Load", "address": 44 },
   { "type": "Store", "vpn": 1, "offset": 1, "data": 999 },
   { "type": "Load", "address": 16 },
   { "type": "Store", "vpn": 3, "offset": 7, "data": 555 },
   { "type": "Load", "address": 48 }
 ]
}
```

Figure 6.2: Configuration file

The simulation is initialized with the following parameters:

- **Virtual Address Width**: 9 bits (512 virtual addresses).
- **Page Size**: 16 bytes (32 virtual pages in total).

- **TLB Size**: 4 entries.
- **Physical Memory Size**: 64 bytes (4 physical pages of 16 bytes each).
- **Secondary Memory Size**: 512 bytes (32 virtual pages of 16 bytes each).
- **Replacement Algorithms Tested**: FIFO, LRU, NRU, Optimal.
- **Future Access Pattern**: [0, 1, 4, 0, 1, 5, 6, 0, 4, 2, 7, 1, 8, 3, 5, 6, 0, 6, 4, 1, 7, 0, 2, 8, 1, 3, 0, 4, 6, 7, 2, 3, 8, 0]
- **Operations**: A series of page allocations, loads, and stores (5 Allocate, 16 Store, and 9 Load operations, executed sequentially).

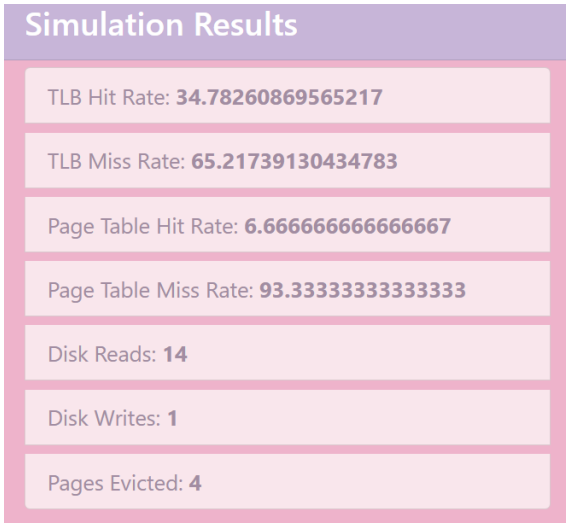


Figure 6.3: LRU Simulation Results

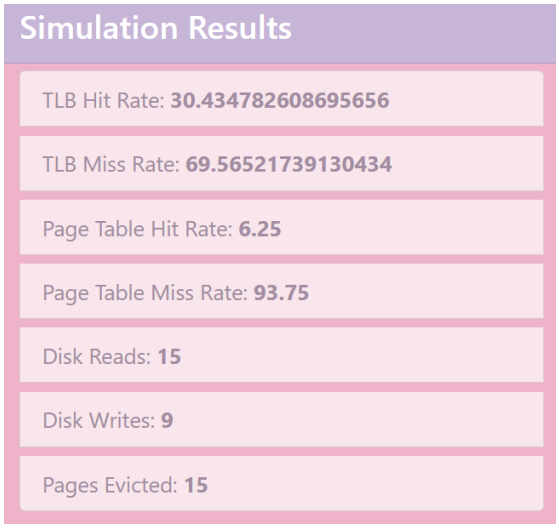


Figure 6.4: FIFO Simulation Results

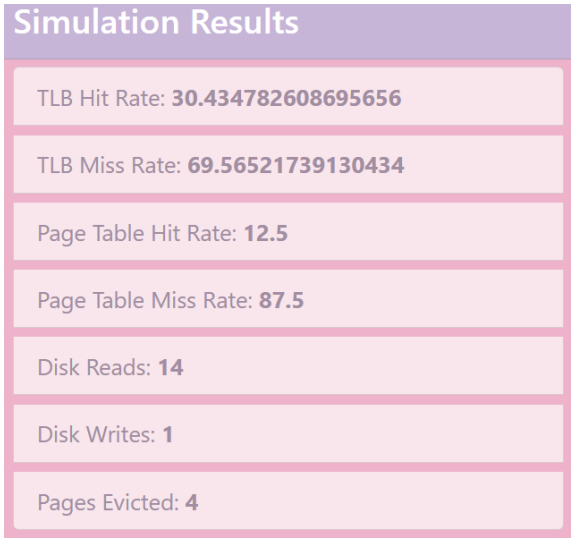


Figure 6.5: NRU Simulation Results

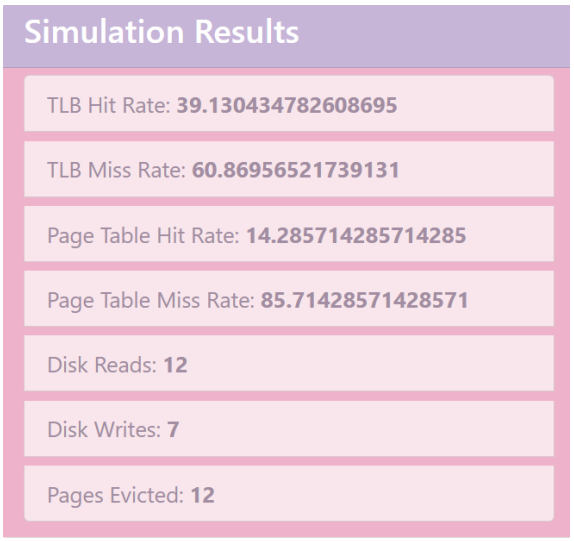


Figure 6.6: Optimal Simulation Results

Initially, virtual pages 0 to 3 are mapped to the 4 physical frames, while virtual page 4 and subsequent pages remain in secondary storage. The

operations are executed sequentially, and the behaviour of the simulator depends on the page replacement algorithm in use.

The first operation after the page allocation is a store at VPN 1 which results in a page table hit, the data being written to the corresponding frame. The second operation is a store at VPN 4, which causes a page fault for all 4 configurations because VPN 4 is not in physical memory. For FIFO, the oldest page in memory (VPN 0) is evicted and VPN 4 is loaded into its frame from the secondary storage. For LRU, the page to be evicted will also be VPN 0, because it is the least recently accessed page. For NRU, the evicted page is VPN 0, being classified as "not recently used" and "not dirty" (Class 0). As for the Optimal replacement, the VPN 3 will be is evicted because it is used farthest in the future (at step 13).

The third operation is a load from the VPN 0. For the Optimal replacement this results in a hit, the data being read from the physical address 12 (frame 0, offset 12). On the other hand, for the FIFO, NRU, and LRU it triggers a page table miss. In the FIFO configuration, VPN 1 will be evicted to make room for VPN 0, while in the LRU configuration VPN 2 will be replaced as it has the oldest access time (3), and in the NRU configuration VPN 3 (class 0) will be replaced.

The fourth operation is a store at VPN 6 which triggers the eviction process in all 4 configurations. For FIFO, VPN 2 will be evicted. For LRU, VPN 3 (with the oldest access time = 4) will be evicted. The Optimal replacement will evict VPN 2 (next use at step 9).

For subsequent operations, FIFO continues to evict the oldest page, causing repeated faults for frequently accessed pages, e.g., VPN 0 and VPN 1 are repeatedly evicted and reloaded. LRU dynamically adjusts to access patterns. VPNs accessed frequently (e.g., VPN 4, VPN 6) remain in memory longer, reducing unnecessary page faults. NRU dynamically categorizes pages, prioritizing eviction of unused and unmodified pages; frequently used or dirty pages are preserved longer. Optimal Replacement minimizes page faults by perfectly predicting future accesses.

Remarks:

- FIFO:
  - simple implementation, requires no additional tracking beyond maintaining a queue of pages
  - inefficient for frequent accesses: frequently used pages are evicted, increasing the number of page faults

    - no adaptation to access patterns, leading to redundant replacements.
- LRU:
    - efficient for workloads with repeated memory accesses
    - minimizes page faults in predictable access patterns
    - requires maintaining access history, introducing overhead
    - may still perform poorly in workloads with highly irregular access patterns.
- NRU:
    - balances eviction decisions based on recent usage and modification status
    - reduces unnecessary disk writes by avoiding eviction of dirty pages
    - performance depends on accurate classification of pages.
- Optimal:
    - produces the lowest number of page faults
    - requires future access knowledge, making it impractical for real systems
    - cannot adapt to changes in access patterns.

Conclusions

At first glance, simulating virtual memory management and handling various memory operations may seem like a straightforward task. However, throughout the analysis, design, and implementation, it becomes clear that the complexity grows as different configurations, scenarios, and algorithms are introduced. Each solution for memory management brings its own set of challenges that must be addressed effectively.

Despite the complexities, the goal of creating a flexible and functional virtual memory simulator has been achieved. The system can now load configurations, allocate virtual pages, map them to physical memory, and handle load/store operations with various page replacement algorithms. With these capabilities, the simulation offers valuable insights into memory management processes.

Moving forward, the simulator can serve as a foundation for further refinement and experimentation, allowing for deeper exploration of memory optimization techniques or expansion to include more advanced features.

Bibliography

[1] Abhishek Bhattacharjee and Daniel Lustig, "Architectural and Operating System Support for Virtual Memory"

[2] David A. Patterson and John L. Hennessy, "Computer organization and design: the hardware/ software interface 5th ed." – chapter 5, pg. 427 - 454

[3] TechTarget - What is Virtual Memory?

[4] Abraham Silberschatz, Peter Baer, Greg Gagne – "Operating System Concepts", ninth edition, chapters 8 & 9

[5] Andrew S. Tanenbaum and Herbert Bos, "Modern Operating Systems" - fourth edition, pg. 181 Live – 263