Department of Computer Science
Technical University of Cluj-Napoca

Virtual Memory Simulator

Name:Stroie Anca Gabriela
Group:30432
Email:Stroie.Io.Anca@student.utcluj.ro

Contents

Chapter 1

Introduction

1.0.1 Project Proposal

The goal of this project is to develop a Java-based virtual memory simulation, providing an educational and experimental platform for understanding the fundamental concepts of virtual memory, memory management, and page-based memory systems.

1.0.2 Project Objectives

Educational Value: Create a user-friendly and interactive simulation program that serves as a valuable learning tool for understanding virtual memory concepts.

Graphical Representation: Develop a visually engaging graphical user interface (GUI) that allows users to visualize key components of virtual memory systems, such as main memory, virtual memory, and the movement of data between them. Use graphical elements to illustrate the concepts and operations involved in virtual memory management.

Key Operations: Implement the fundamental virtual memory operations:

- Page allocation: Demonstrating the process of loading pages from virtual memory into main memory.
- Page replacement: Showing how the system selects pages for replacement when main memory is full.
- Page table management: Visualizing the mapping of virtual addresses to physical addresses.

Configurability: Provide users with the ability to customize simulation parameters, including:

- Memory size: Allow users to set the size of both main memory and virtual memory.
- Page size: Permit users to configure the size of memory pages.

1.0.3 Project Components

1. Graphical User Interface (GUI): Develop a user-friendly interface that visually represents the main memory, virtual memory, and page operations. Allow users to interact with the simulation.

2. Simulation Logic: Implement the core logic for virtual memory operations, including page allocation, page replacement, and page table management.

3. Configuration Options: Create settings that enable users to customize simulation parameters, such as memory size, page size, and page replacement algorithms.

4. User Documentation: Prepare user documentation, including a user guide and tutorial materials that explain how to use the simulation effectively.

1.0.4 Weekly Plan

1. Week 1: Project initiation, research, and proposal development.

2. Week 2: GUI design and development, focusing on visual representation and user interactivity.

3. Week 3: Implementation of simulation logic, covering page allocation, page replacement, and page table management.

4. Week 4: Configuration options and customization, ensuring flexibility and user configurability.

5. Week 5: User documentation preparation, creating clear and informative guides and tutorials.

6. Week 6: User testing and feedback collection to evaluate usability and functionality.

7. Week 7: Final review, debugging, and presenting the project.

Chapter 2

Bibliographic Study

2.1 What is Virtual Memory

Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of the main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites and program-generated addresses are translated automatically to the corresponding machine addresses. The computer's operating system, using a combination of hardware and software, maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory. Main storage, as seen by a process or task, appears as a contiguous address space or collection of contiguous segments. The operating system manages virtual address spaces and the assignment of real memory to virtual memory. Address translation hardware in the CPU, often referred to as a memory management unit (MMU), automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities, utilizing, e.g., disk storage, to provide a virtual address space that can exceed the capacity of real memory and thus reference more memory than is physically present in the computer.

2.2 Paging

Paging is a virtual memory technique that separates memory into sections called paging files. When a computer reaches its RAM limits, it transfers any currently unused pages into the part of its hard drive used for virtual memory. The computer performs this process using a swap file, a designated space within its hard drive for extending the virtual memory of the computer's RAM. By moving unused files into its hard drive, the computer frees its RAM space for other memory tasks and ensures that it doesn't run out of real memory.

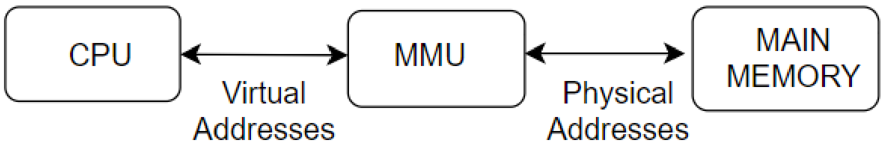


Figure 2.1: Virtual Memory

As part of this process, the computer uses page tables, which translate virtual addresses into the physical addresses that the computer's memory management unit (MMU) uses to process instructions. The MMU communicates between the computer's OS and its page tables. When the user performs a task, the OS searches its RAM for the processes to conduct the task. If it can't find the processes to complete the task in RAM, the MMU prompts the OS to move the required pages into RAM and uses a page table to note the new storage location of the pages.

Chapter 3

Analysis

The primary objective of this application is to provide a user-friendly virtual memory simulation environment for educational and learning purposes. It should help users understand the concepts of virtual memory, page management, and page replacement algorithms. This tool can be targeted at students, educators, and anyone interested in understanding how modern operating systems manage memory.

3.1 Requirements

- Simulate Virtual Memory: The application should accurately simulate virtual memory systems with the ability to load processes into memory.
- Page Replacement Algorithms: Support various page replacement algorithms like FIFO.
- User Interface: Develop an intuitive and interactive user interface that allows users to configure simulations, view real-time statistics, and analyze results.
- Performance Metrics: Track and display performance metrics such as page hit rate, page fault rate, and memory usage.
- Customization: Allow users to configure parameters like memory size, page size, and the number of processes.
- Extensibility: Provide an option for users to create custom page replacement algorithms.
- Error Handling: Implement informative error messages and recovery mechanisms for scenarios like invalid inputs or system crashes.

3.2 Scenarios

1. Loading a Process
    - A user opens the application.
    - They select the option to load a process into virtual memory.
    - The user chooses a process file and specifies its size.
    - The simulator loads the process, assigns page frames, and updates the page table.

2. Page Replacement

- The user configures the simulator with a specific page replacement algorithm (e.g FIFO).
- Runs a simulation with multiple processes, causing page faults.
- The simulator demonstrates how pages are replaced based on the chosen algorithm.

3. Monitoring Performance

- During a simulation, the user can monitor various performance metrics, such as page hit rate, page fault rate, and memory usage.
- The simulator provides real-time performance statistics.

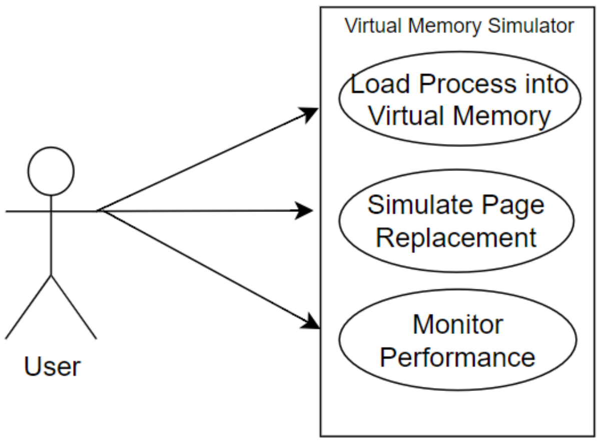


Figure 3.1: Use Cases

3.3 Algorithms

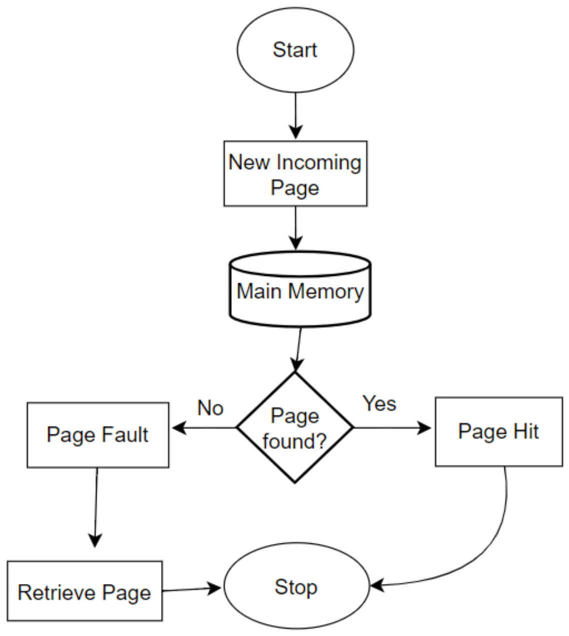
3.3.1 FIFO

Page replacement algorithms come under the paging technique. Paging is a method that allows a computer to retrieve and store data from the secondary memory (e.g. HDD or drum memory) into the main memory (RAM).Page replacement algorithms like FIFO are used when there is a new page request, and there is not enough space in the main memory to allocate the new page.

Hence, a page replacement algorithm decides which page it should replace so that it can allocate the memory for the new page. The steps of a page replacement can be summarized in a flowchart(fig 3.2). First-in, first-out (FIFO) algorithm has a simple approach to this problem. We keep track of all the pages by using a queue in the main memory. As soon as a page comes in, we'll put it in the queue and continue. In this way, the oldest page would always be in the first place of the queue.

Now when a new page comes in and there is no space in the memory, we remove the first page in the queue, which is also the oldest one. It repeats this process until the operating system has page flow in the system.

The main advantage of the FIFO page replacement algorithm is its simplicity. It is easy to understand and implement. It also uses a queue data structure.



figureFlow Chart Diagram

Chapter 4

Design

The programm will implement the MVC architectural pattern to separate the simulator into three main components: Model, View, and Controller.

Model: Represents the data and core simulation logic. This includes the memory management unit (MMU), page table, and page replacement algorithms.

View: Provides the user interface elements for interaction, including buttons, input fields, and real-time graphical displays of memory utilization and statistics.

Controller: Acts as an intermediary that processes user inputs from the View and communicates with the Model to trigger simulations and updates.

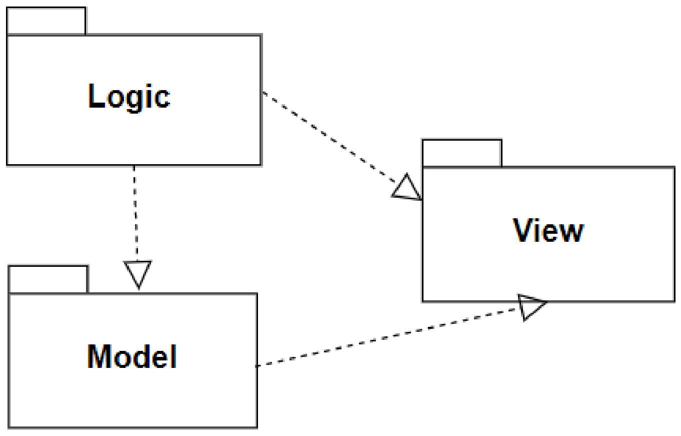


Figure 4.1: Package diagrame

4.1 User Interface Components

1. TLB Table
    - The TLB table is a component of the virtual memory management system that caches translations of virtual addresses to physical addresses, reducing the overhead of page table lookups.

- Each TLB entry contains a tag field and a data field. The tag field holds the virtual address, while the data field stores the corresponding physical address and additional control bits or information.

2. Page Table

   - The page table is a critical component that maps virtual page numbers to physical page frame numbers. It plays a central role in the address translation process

3. Physical Memory Table

   - The physical memory table reflects the actual memory space where page frames are allocated to processes. It provides users with insights into the allocation of physical pages.

4. Slots for memory size and page size

   - Allow users to configure the size of the virtual memory and the page size. These parameters significantly affect how the simulator operates.

4.2 Memory Management Unit

The MMU is a fundamental part of the "Model." It serves as the bridge between the CPU and memory, responsible for translating virtual addresses to physical addresses and managing memory-related operations.

Functions:

- **Address Translation:** The MMU handles the translation of virtual addresses to physical addresses using the page table or other data structures.
- **Page Fault Handling:** It manages page faults by bringing required pages from secondary storage into physical memory.
- **TLB Interaction:** The MMU interacts with the Translation Lookaside Buffer (TLB) to expedite address translation and improve performance.

How it works?

1. The requested address is broken down into a page and an offset.
2. The requested page is searched in the TLB with a fully associative principle.
3. If there is a frame number that matches the page, a TLB HIT is obtained, and data will be loaded from the TLB.
4. If the page is not found in the TLB, the page will be searched through the page table with a direct mapping principle.
5. If a matching page ID is found in the page table, a Page Table Hit is obtained, and data will be loaded from the page table. The TLB will be updated accordingly.
6. If neither the TLB nor the page table contains the required page, a "Miss" is obtained, and data will be loaded from secondary memory.

Chapter 5

Implementation

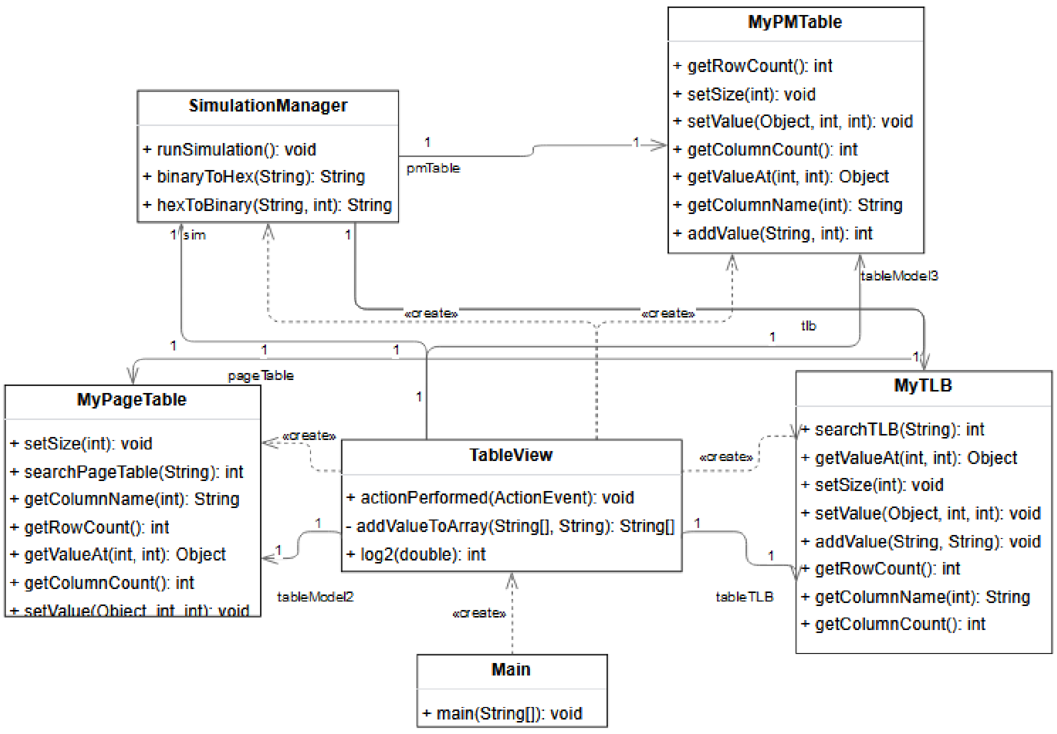

MyPMTable
+ getRowCount(): int
+ setSize(int): void
+ setValue(Object, int, int): void
+ getColumnCount(): int
+ getValueAt(int, int): Object
+ getColumnName(int): String
+ addValue(String, int): int
SimulationManager
+ runSimulation(): void
+ binaryToHex(String): String
+ hexToBinary(String, int): String
1
1
pmTable
1
sim
1
tableModel3
«create»
«create»
tlb
1
1
1
1
1
pageTable
1
MyPageTable
+ setSize(int): void
+ searchPageTable(String): int
+ getColumnName(int): String
+ getRowCount(): int
+ getValueAt(int, int): Object
+ getColumnCount(): int
+ setValue(Object, int, int): void
«create»
TableView
+ actionPerformed(ActionEvent): void
- addValueToArray(String[], String): String[]
+ log2(double): int
«create»
MyTLB
+ searchTLB(String): int
+ getValueAt(int, int): Object
+ setSize(int): void
+ setValue(Object, int, int): void
+ addValue(String, String): void
+ getRowCount(): int
+ getColumnName(int): String
+ getColumnCount(): int
1
1
1
1
tableModel2
tableTLB
«create»
Main
+ main(String[]): void

Figure 5.1: Class Diagram

The application is structured into three main packages:

- **org.example.model** Contains the data models representing different components of a virtual memory system such as MyPageTable, MyPMTable, and MyTLB.
- **org.example.logic** Contains the data models representing different components of a virtual memory system such as MyPageTable, MyPMTable, and MyTLB.
- **org.example.view** Includes TableView, a GUI component built using Swing, providing an interactive interface for the simulation.

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's a detailed look at each class within the model package:

5.1.1 MyPageTable

The MyPageTable class in the org.example.model package is a Java class that models the concept of a page table in a virtual memory system. It extends AbstractTableModel, which is a Swing class used to define the data model for a table component. Let's break down its structure and functionality:

Fields:table-a two-dimensional String array that represents the data structure for the page table entries.

Methods:

- getRowCount()-returns the number of rows in the table, which corresponds to the length of the table array.
- getColumnCount()-returns the fixed number of columns in the table, which is 3 in this implementation.
- getValueAt(int rowIndex, int columnIndex)-retrieves the value at the specified row and column index. It checks if the indices are within the bounds of the table before retrieving the value.
- getColumnName(int c)-returns the name of the column at the specified index. The names are "Index", "Valid", and "Physical Page" for columns 0, 1, and 2, respectively.
- setValue(Object s, int r, int c)-sets the value at the specified row (r) and column (c) to the value s. If the indices are within the bounds of the table, the value is converted to a String and stored. After setting the value, it notifies all listeners that the table's data has changed by calling 'fireTableDataChanged
- setSize(int rowCount)-adjusts the size of the page table to the specified number of rows. It maintains the number of columns at 3. The method also initializes the "Index" column with hexadecimal values of the row indices, and sets the "Valid" column to "0" for each row, indicating that the pages are initially invalid. The remaining columns are filled with empty strings. After setting up the new table, it calls fireTableDataChanged() to update any views attached to this model.

Functionality:

The MyPageTable class models a simple page table for use in a memory management simulation. A page table's primary function is to map virtual pages to frames in physical memory. In this simulation:

Each row represents a virtual page entry. The first column, "Index", holds the virtual page number. The second column, "Valid", indicates whether the virtual page is present in physical memory (1 for present, 0 for not present). The third column, "Physical Page", stores the corresponding physical page number if the virtual page is mapped to physical memory.

5.1.2 MyPMTable

The MyPMTable class in the org.example.model package is a Java class that models the concept of a page table in a virtual memory system. It extends AbstractTableModel, which is a Swing class used to define the data model for a table component. Let's break down its structure and functionality: **Fields:**table: A two-dimensional String array representing the contents of physical memory. Each row corresponds to a physical memory block or page, and there are two columns to store information about each block.

index: An integer index used to keep track of the next available row in the physical memory table for adding new entries.

Methods:

- getRowCount()-returns the number of rows in the table, which corresponds to the length of the table array.
- getColumnCount()-returns the fixed number of columns in the table, which is 3 in this implementation.
- getValueAt(int rowIndex, int columnIndex)-retrieves the value at the specified row and column index. It checks if the indices are within the bounds of the table before retrieving the value.
- getColumnName(int c)-returns the name of the column at the specified index. The names are "Index", "Valid", and "Physical Page" for columns 0, 1, and 2, respectively.
- setValue(Object s, int r, int c)-sets the value at the specified row (r) and column (c) to the value s. If the indices are within the bounds of the table, the value is converted to a String and stored. After setting the value, it notifies all listeners that the table's data has changed by calling 'fireTableDataChanged
- getColumnName(int c)-provides names for the columns based on the index. The first column (index 0) is named "Physical Page", and the second column (index 1) is named "Content". These names are likely used by the JTable GUI component to display headers.
- setSize(int rowCount)-adjusts the size of the table to accommodate a new number of rows, maintaining the column count at 2. It initializes the "Physical Page" column with hexadecimal strings representing the row indices. This method is likely used to configure the table size based on simulation parameters.

Functionality:

The MyPMTable class is responsible for managing the representation of physical memory in the simulation:

The "Physical Page" column likely represents the frame number or the index of the physical memory block. The "Content" column contains the actual content of the physical memory block, which in this case is a string representation of the data stored, including the starting and ending word indices within the block.

5.1.3 MyTLB

The MyTLB class represents a Translation Lookaside Buffer (TLB) within a virtual memory system. It extends AbstractTableModel from the Swing library, allowing it to be used as the data model for a JTable component to visually represent the TLB in a GUI.

The TLB is an essential component in a virtual memory system, providing fast access to the physical addresses corresponding to recently accessed virtual pages. The MyTLB class captures this functionality in a simulated environment, allowing the virtual memory system to demonstrate how a TLB operates within the larger context of address translation. The JTable GUI component can visualize this operation using the MyTLB class as its data model, showing TLB hits and misses and the current state of the TLB after each memory access or context switch.

5.2 Logic Package

5.2.1 SimulationManager

The SimulationManager class in the org.example.logic package is responsible for managing the logic of a virtual memory simulation. This class interacts with model classes MyTLB, MyPMTable, and MyPageTable to simulate the translation of virtual memory addresses to physical addresses and the use of a TLB. Here's a detailed explanation of the code:

- physicalPageSize: An integer to represent the size of the physical memory pages.
- tlbSize: An integer indicating the number of entries in the TLB.
- addressLength: An integer specifying the length of the memory addresses used in the simulation.
- instructionArray: An array of strings, where each string is an instruction that likely contains a memory address to be translated.
- tlb, pmTable, pageTable: Instances of the model classes representing the TLB, the physical memory table, and the page table, respectively.

Methods

- hexToBinary(String hex, int binaryLength): Converts a hexadecimal string to a binary string. The method uses a predefined map to translate each hexadecimal digit into its binary equivalent. It handles padding the result with leading zeros to match the specified binaryLength or truncates excess leading bits if necessary. It also checks for invalid hexadecimal characters and throws an IllegalArgumentException if any are found.
- runSimulation(): This method simulates the processing of instructions. It prints out the binary representation of each instruction in the instructionArray, converting them from hexadecimal to binary using the hexToBinary method. This simulates a part of the address translation process where the CPU-generated virtual addresses (in hexadecimal) are converted to binary for further processing in memory management.
- binaryToHex(String s): Converts a binary string back to a hexadecimal string. This might simulate the reverse process of taking a binary memory address and converting it back to a format that is easier to display or interpret.

Functionality The SimulationManager acts as a controller that orchestrates the virtual memory simulation. It primarily:

Manages the conversion between hexadecimal and binary formats, which is a common operation in memory address translation. Drives the simulation by processing an array of memory access instructions, which might involve further operations not detailed in the provided code, such as updating the TLB (MyTLB), physical memory table (MyPMTable), and page table (MyPageTable) based on memory access outcomes (page hits or misses). Ensures the integrity of data by checking for valid hexadecimal and binary strings, throwing exceptions when invalid data is encountered.

5.3 View Package

The TableView class, part of the org.example.view package, acts as the user interface for a virtual memory simulator application. It is a JFrame subclass, which is a top-level container provided by Java Swing to represent a window on the screen. The class also implements the ActionListener interface, enabling it to respond to user actions like button clicks.

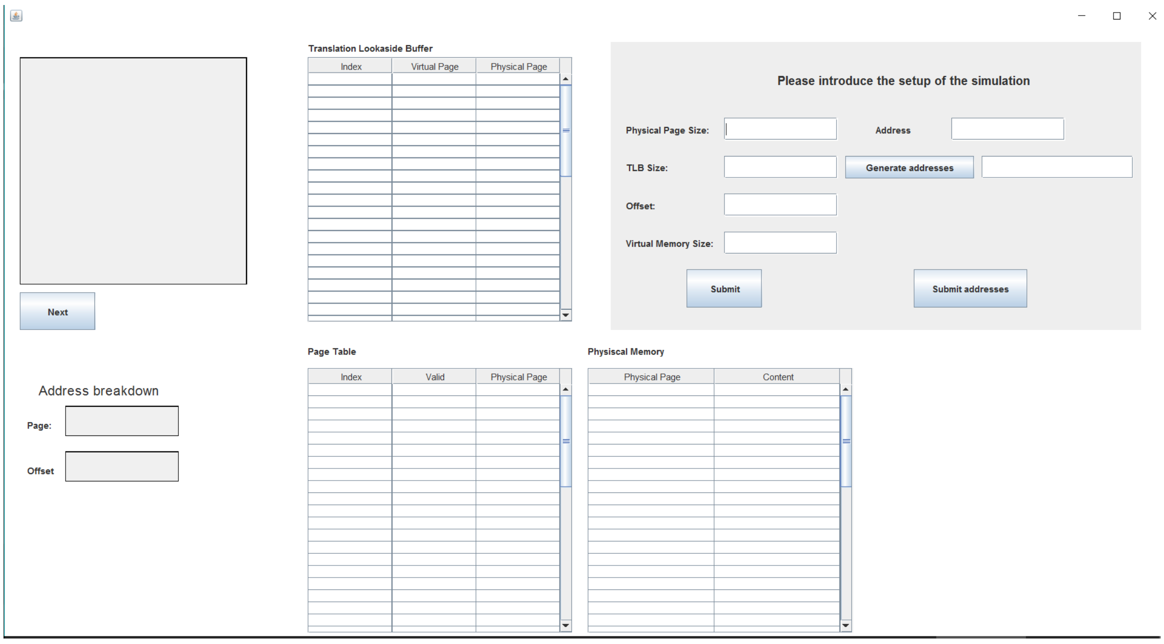


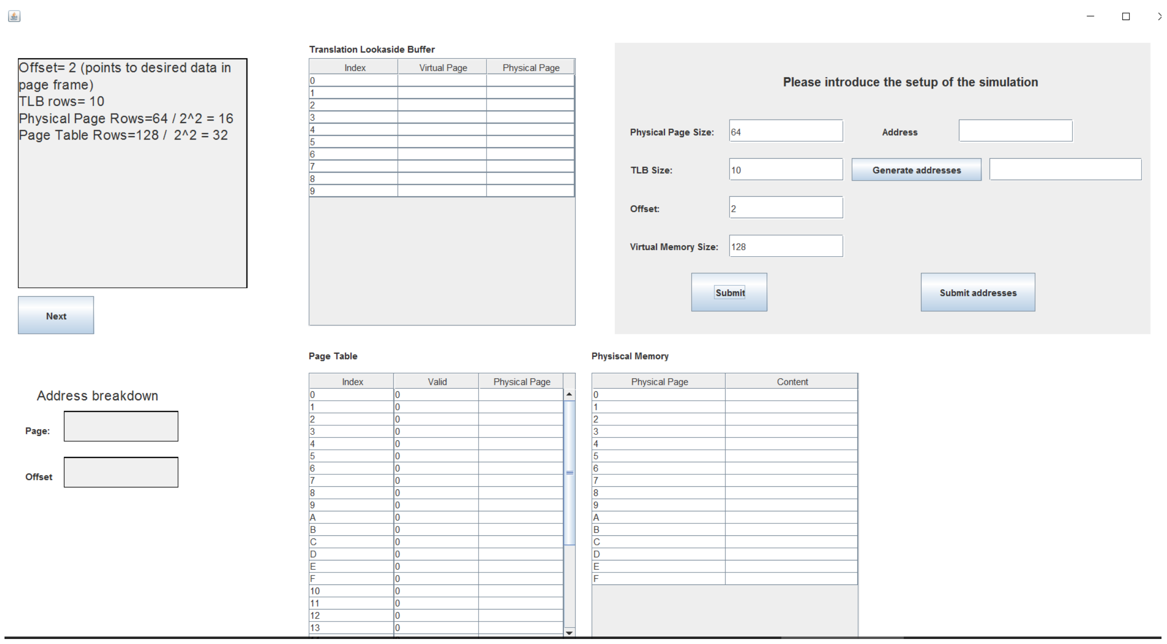
Figure 5.2: Graphical User Interface

Chapter 6

Tests/Conclusions

In the interface, the user can introduce the setup of the simulation: the size of the physical page, the size of the TLB, the size of the offset, the size of the virtual memory and the array of addresses.

First the user must introduce the values. The tables will be updated according to the given sizes.



Offset= 2 (points to desired data in page frame)
TLB rows= 10
Physical Page Rows=64 / 2^2 = 16
Page Table Rows=128 / 2^2 = 32

Translation Lookaside Buffer

Index	Virtual Page	Physical Page
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Please introduce the setup of the simulation

Physical Page Size: 64
Address:
TLB Size: 10
Offset: 2
Virtual Memory Size: 128

Address breakdown
Page:
Offset

Page Table

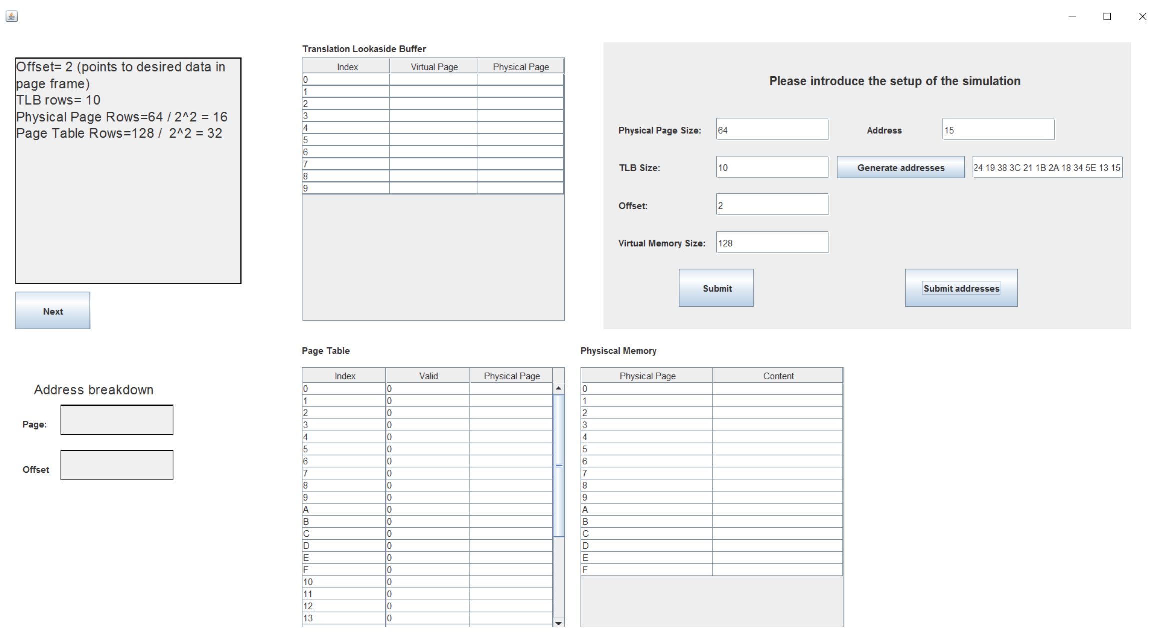
Index	Valid	Physical Page
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
7	0	
8	0	
9	0	
A	0	
B	0	
C	0	
D	0	
E	0	
F	0	
10	0	
11	0	
12	0	
13	0	

Physiscal Memory

Physical Page	Content
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
A	
B	
C	
D	
E	
F	

Figure 6.1: First step of simulation

The button Submit is pressed . Then the user introduce the array of addresses: 15 75 1E 24 19 38 3C 21 1B 2A 18 34 5E 13 15. Then the button Submit addresses is pressed.



Offset= 2 (points to desired data in page frame)
TLB rows= 10
Physical Page Rows=64 / 2^2 = 16
Page Table Rows=128 / 2^2 = 32

Next

Address breakdown

Page:

Offset

Translation Lookaside Buffer

Index	Virtual Page	Physical Page
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Please introduce the setup of the simulation

Physical Page Size: 64
Address: 15
TLB Size: 10
Generate addresses: 24 19 38 3C 21 1B 2A 18 34 5E 13 15
Offset: 2
Virtual Memory Size: 128

Submit | Submit addresses

Page Table

Index	Valid	Physical Page
0	0	
1	0	
2	0	
3	0	
4	0	
5	0	
6	0	
7	0	
8	0	
9	0	
A	0	
B	0	
C	0	
D	0	
E	0	
F	0	
10	0	
11	0	
12	0	
13	0	

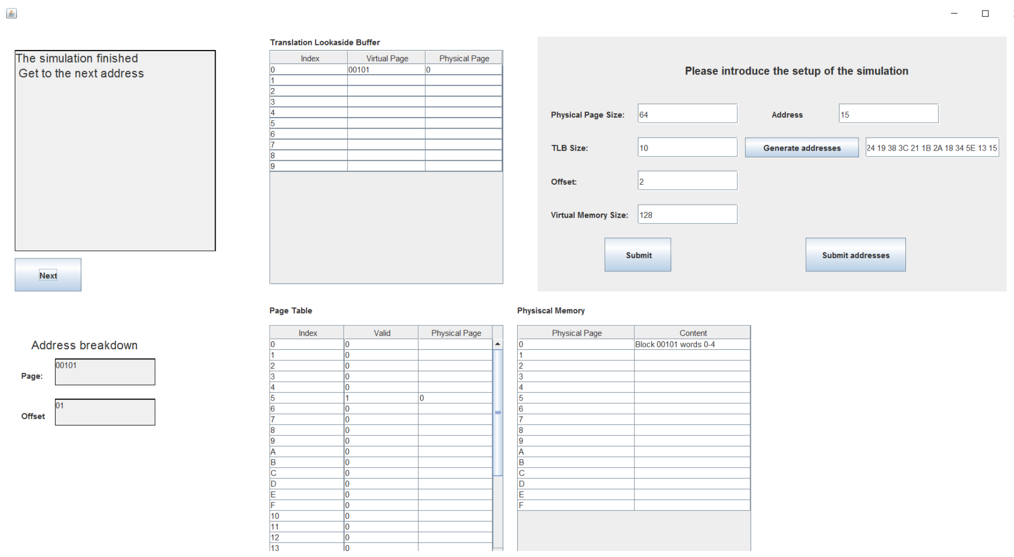
Physiscal Memory

Physical Page	Content
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
A	
B	
C	
D	
E	
F	

Figure 6.2: Second step of simulation

Forward, we can see how the virtual memory works for every address in the array by pressing the button Next. A number of steps will then be executed by pressing Next.

- first the address will be broken down into page and offset.
- then the requested page is searched in TLB.
- Page Hit or Miss will be displayed. If it is page Hit, data will be loaded from TLB and the simulation finished, else continue with the steps.
- the requested page will be searched in Page Table
- Page Table Hit or Miss will be displayed. If it is a Hit, data will be loaded from Page Table, else data will be uploaded from Secondary Memory and TLB, Page Table, and Physical Memory will be updated
- get to the next address



The simulation finished
Get to the next address

Next

Address breakdown

Page: 00101

Offset: 01

Translation Lookaside Buffer

Index	Virtual Page	Physical Page
0	00101	0
1		
2		
3		
4		
5		
6		
7		
8		
9		

Please introduce the setup of the simulation

Physical Page Size: 64
Address: 15
TLB Size: 10
Generate addresses: 24 19 38 3C 21 1B 2A 18 34 5E 13 15
Offset: 2
Virtual Memory Size: 128

Submit | Submit addresses

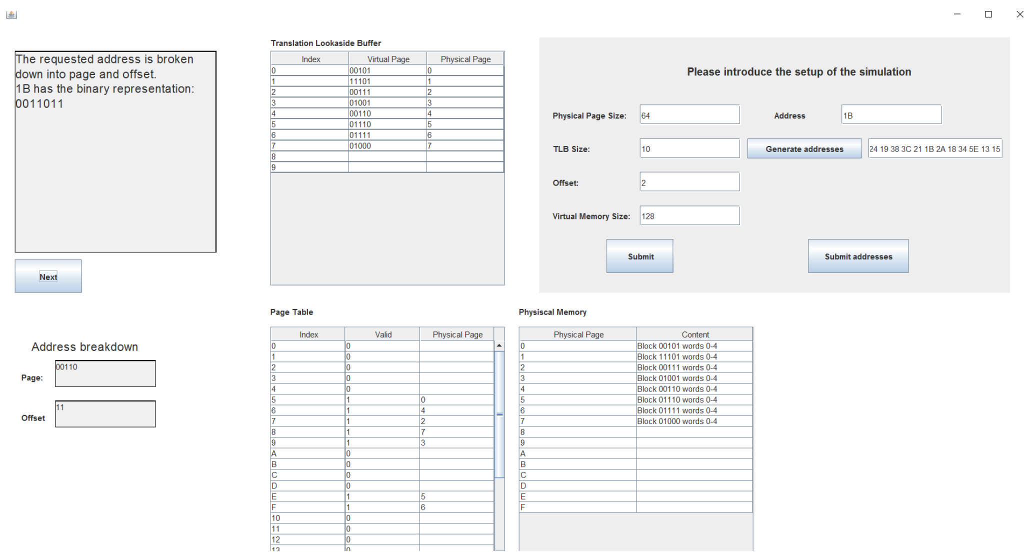
Page Table

Index	Valid	Physical Page
0	0	
1	0	
2	0	
3	0	
4	0	
5	1	0
6	0	
7	0	
8	0	
9	0	
A	0	
B	0	
C	0	
D	0	
E	0	
F	0	
10	0	
11	0	
12	0	
13	0	

Physiscal Memory

Physical Page	Content
0	Block 00101 words 0-4
1	
2	
3	
4	
5	
6	
7	
8	
9	
A	
B	
C	
D	
E	
F	

Figure 6.3: Simulation for the first address



The requested address is broken down into page and offset.
1B has the binary representation: 0011011

Address breakdown

Page: 00110

Offset 11

Translation Lookaside Buffer

Index	Virtual Page	Physical Page
0	00101	0
1	11101	1
2	00111	2
3	01001	3
4	00110	4
5	01110	5
6	01111	6
7	01000	7
8		
9		

Please introduce the setup of the simulation

Physical Page Size: 64

Address 1B

TLB Size: 10

24 19 38 3C 21 1B 2A 18 34 5E 13 15

Offset: 2

Virtual Memory Size: 128

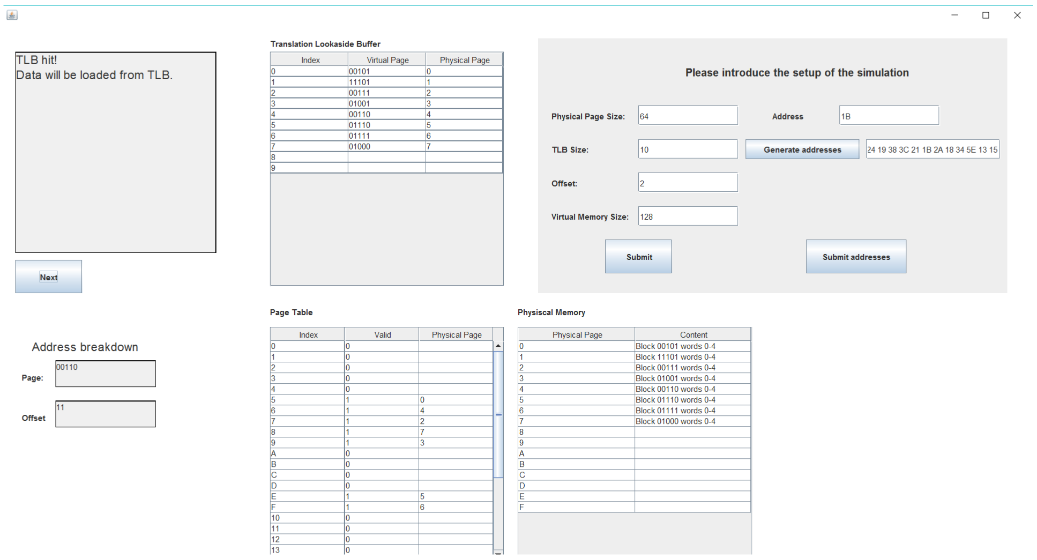
Page Table

Index	Valid	Physical Page
0	0	
1	0	
2	0	
3	0	
4	0	
5	1	0
6	1	4
7	1	2
8	1	7
9	1	3
A	0	
B	0	
C	0	
D	0	
E	1	5
F	1	6
10	0	
11	0	
12	0	
13	0	

Physiscal Memory

Physical Page	Content
0	Block 00101 words 0-4
1	Block 11101 words 0-4
2	Block 00111 words 0-4
3	Block 01001 words 0-4
4	Block 00110 words 0-4
5	Block 01110 words 0-4
6	Block 01111 words 0-4
7	Block 01000 words 0-4
8	
9	
A	
B	
C	
D	
E	
F	

Figure 6.4: Simulation for 1B



TLB hit!
Data will be loaded from TLB.

Address breakdown

Page: 00110

Offset 11

Translation Lookaside Buffer

Index	Virtual Page	Physical Page
0	00101	0
1	11101	1
2	00111	2
3	01001	3
4	00110	4
5	01110	5
6	01111	6
7	01000	7
8		
9		

Please introduce the setup of the simulation

Physical Page Size: 64

Address 1B

TLB Size: 10

24 19 38 3C 21 1B 2A 18 34 5E 13 15

Offset: 2

Virtual Memory Size: 128

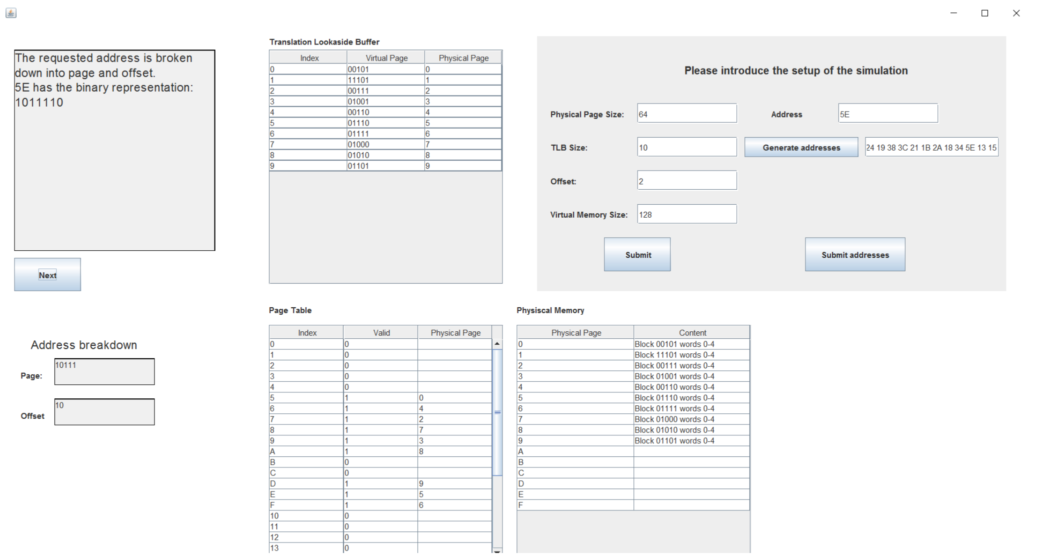
Page Table

Index	Valid	Physical Page
0	0	
1	0	
2	0	
3	0	
4	0	
5	1	0
6	1	4
7	1	2
8	1	7
9	1	3
A	0	
B	0	
C	0	
D	0	
E	1	5
F	1	6
10	0	
11	0	
12	0	
13	0	

Physiscal Memory

Physical Page	Content
0	Block 00101 words 0-4
1	Block 11101 words 0-4
2	Block 00111 words 0-4
3	Block 01001 words 0-4
4	Block 00110 words 0-4
5	Block 01110 words 0-4
6	Block 01111 words 0-4
7	Block 01000 words 0-4
8	
9	
A	
B	
C	
D	
E	
F	

Figure 6.5: For 1B it will be a TLB hit



The requested address is broken down into page and offset.
5E has the binary representation:
1011110

Next

Address breakdown

Page: 10111

Offset: 10

Translation Lookaside Buffer

Index	Virtual Page	Physical Page
0	00101	0
1	11101	1
2	00111	2
3	01001	3
4	00110	4
5	01110	5
6	01111	6
7	01000	7
8	01010	8
9	01101	9

Please introduce the setup of the simulation

Physical Page Size: 64

Address: 5E

TLB Size: 10

Generate addresses: 24 19 38 3C 21 1B 2A 18 34 5E 13 15

Offset: 2

Virtual Memory Size: 128

Submit

Submit addresses

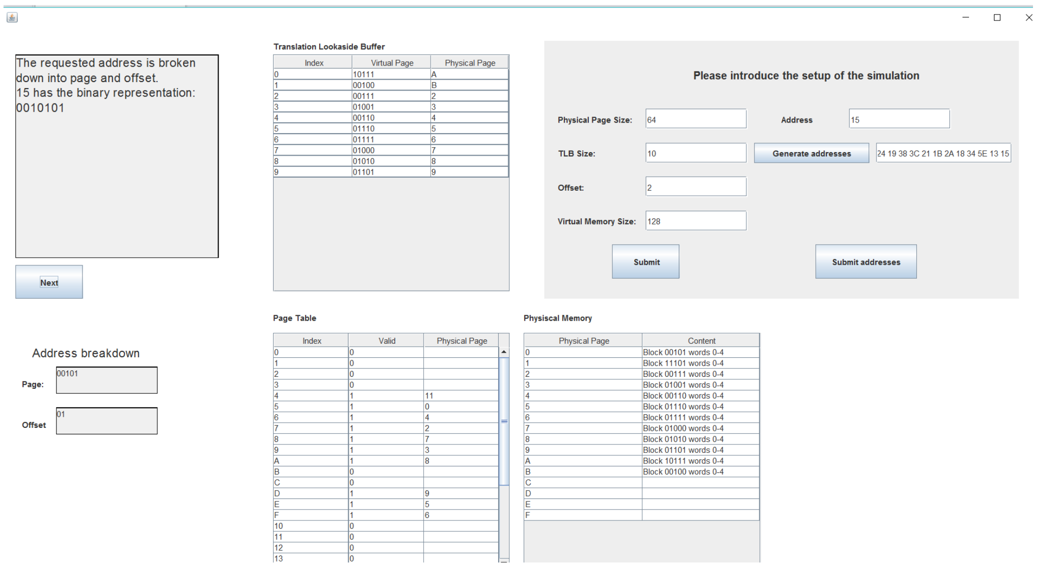
Page Table

Index	Valid	Physical Page
0	0	
1	0	
2	0	
3	0	
4	0	
5	1	0
6	1	4
7	1	2
8	1	7
9	1	3
A	1	8
B	0	
C	0	
D	1	9
E	1	5
F	1	6
10	0	
11	0	
12	0	
13	0	

Physiscal Memory

Physical Page	Content
0	Block 00101 words 0-4
1	Block 11101 words 0-4
2	Block 00111 words 0-4
3	Block 01001 words 0-4
4	Block 00110 words 0-4
5	Block 01110 words 0-4
6	Block 01111 words 0-4
7	Block 01000 words 0-4
8	Block 01010 words 0-4
9	Block 01101 words 0-4
A	
B	
C	
D	
E	
F	

Figure 6.6: Now that the TLB is full, addresses will be replaced using FIFO



The requested address is broken down into page and offset.
15 has the binary representation:
0010101

Next

Address breakdown

Page: 00101

Offset: 01

Translation Lookaside Buffer

Index	Virtual Page	Physical Page
0	10111	A
1	00100	B
2	00111	2
3	01001	3
4	00110	4
5	01110	5
6	01111	6
7	01000	7
8	01010	8
9	01101	9

Please introduce the setup of the simulation

Physical Page Size: 64

Address: 15

TLB Size: 10

Generate addresses: 24 19 38 3C 21 1B 2A 18 34 5E 13 15

Offset: 2

Virtual Memory Size: 128

Submit

Submit addresses

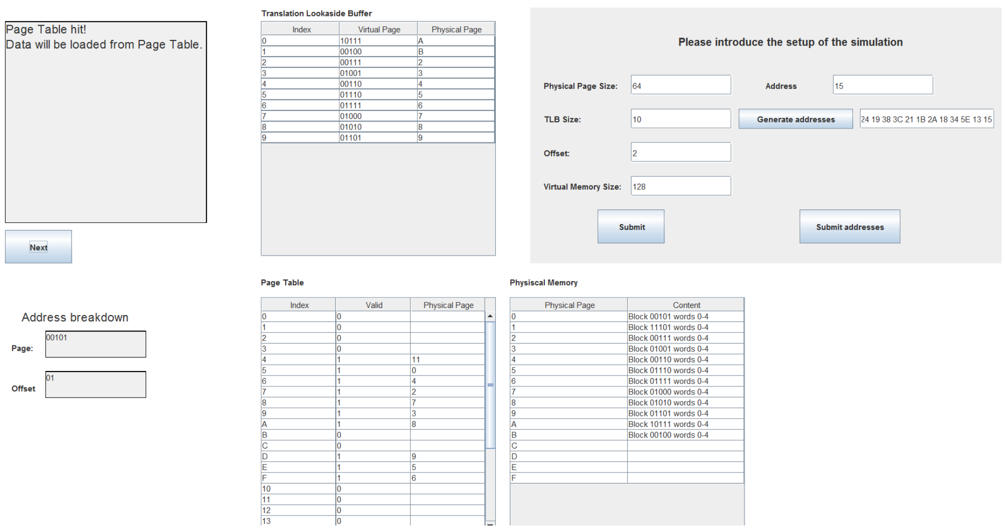
Page Table

Index	Valid	Physical Page
0	0	
1	0	
2	0	
3	0	
4	1	11
5	1	0
6	1	4
7	1	2
8	1	7
9	1	3
A	1	8
B	0	
C	0	
D	1	9
E	1	5
F	1	6
10	0	
11	0	
12	0	
13	0	

Physiscal Memory

Physical Page	Content
0	Block 00101 words 0-4
1	Block 11101 words 0-4
2	Block 00111 words 0-4
3	Block 01001 words 0-4
4	Block 00110 words 0-4
5	Block 01110 words 0-4
6	Block 01111 words 0-4
7	Block 01000 words 0-4
8	Block 01010 words 0-4
9	Block 01101 words 0-4
A	Block 10111 words 0-4
B	Block 00100 words 0-4
C	
D	
E	
F	

Figure 6.7: 15 will be a TLB miss and will be found in Page Table


Page Table hit!
Data will be loaded from Page Table.

Address breakdown

Page: 00101

Offset 01

Translation Lookaside Buffer

Index	Virtual Page	Physical Page
0	10111	A
1	00100	B
2	00111	2
3	01001	3
4	00110	4
5	01110	5
6	01111	6
7	01000	7
8	01010	8
9	01101	9

Please introduce the setup of the simulation

Physical Page Size: 64
Address 15
TLB Size: 10
24 19 38 3C 21 1B 2A 18 34 5E 13 15
Offset: 2
Virtual Memory Size: 128

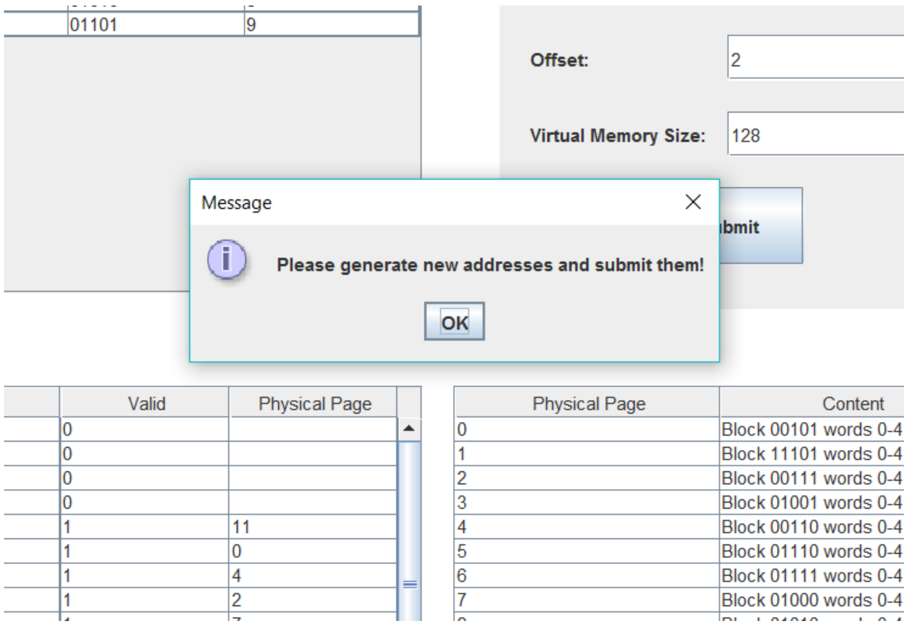
Page Table

Index	Valid	Physical Page
0	0	
1	0	
2	0	
3	0	
4	1	11
5	1	0
6	1	4
7	1	2
8	1	7
9	1	3
A	1	8
B	0	
C	0	
D	1	9
E	1	5
F	1	6
10	0	
11	0	
12	0	
13	0	

Physiscal Memory

Physical Page	Content
0	Block 00101 words 0-4
1	Block 11101 words 0-4
2	Block 00111 words 0-4
3	Block 01001 words 0-4
4	Block 00110 words 0-4
5	Block 01110 words 0-4
6	Block 01111 words 0-4
7	Block 01000 words 0-4
8	Block 01010 words 0-4
9	Block 01101 words 0-4
A	Block 10111 words 0-4
B	Block 00100 words 0-4
C	
D	
E	
F	

Figure 6.8: Page Table Hit.


01101	9

Offset: 2
Virtual Memory Size: 128

Message

Please generate new addresses and submit them!

Valid	Physical Page
0	
0	
0	
0	
1	11
1	0
1	4
1	2

Physical Page	Content
0	Block 00101 words 0-4
1	Block 11101 words 0-4
2	Block 00111 words 0-4
3	Block 01001 words 0-4
4	Block 00110 words 0-4
5	Block 01110 words 0-4
6	Block 01111 words 0-4
7	Block 01000 words 0-4

Figure 6.9: At the end of the array this message will appear

This project aimed to create a virtual memory simulator to help explain how virtual memory works in computers. The simulator was designed to show key processes like how pages are managed, what happens when a page is found or not found (hits and misses), and how the Translation Lookaside Buffer (TLB) and page table work.

The simulator was successful in clearly showing how virtual memory operates. It made complex concepts easier to understand by visually demonstrating them. This was particularly helpful for learning about the various parts of memory management and how they interact.

The simulator did a great job with basic concepts, but there's still room for more. Future versions could include more details about advanced memory management techniques. Another improvement could be to make the simulator more interactive, allowing users to change settings and see what happens.

Bibliography

[1] ”Virtual memory” [Online]. Available: https://en.wikipedia.org/wiki/Virtual_memory

[2] ”Virtual Memory in Operating Systems” [Online]. Available: https://www.geeksforgeeks.org/virtual-memory-in-operating-system/

[3] ”What is a Virtual Memory in OS” [Online]. Available: :https://www.javatpoint.com/os-virtual-memory

[4] Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, ”Operating System Concepts, Ninth Edition ” [Online]. Available: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/9_VirtualMemory.html

[5] ”Operating System-Virtual Memory” [Online]. Available: :https://www.tutorialspoint.com/operating_system/os_virtual_memory.htm