ASSIGNMENT

24MCAT103-Digital Fundamentals and Computer Architecture

24MCAT107-Advanced Software Engineering

Submitted by: vismaya koorma

Roll no:59

RMCA S1

Simulating Cache Memory Mapping and Replacement Algorithms using Git Collaboration

Scenario: You are part of a development team tasked with creating a Cache Simulation Tool that models the behaviour of different cache mapping functions (Direct, Associative, Set Associative) and cache replacement algorithms (FIFO, LRU, LFU).

Part A – 24MCAT103-Digital Fundamentals and Computer Architecture Component (Simulation Design)

PROGRAM

```
def lru(pages, frames):
    memory = []
    page_faults = 0

    print("\n--- LRU Page Replacement ---")
    for p in pages:

        # Case 1: Page Hit
        if p in memory:
            memory.remove(p)
            memory.append(p)

        else:
            # Page Fault
            page_faults += 1
```

```
            # If space available, add page
            if len(memory) < frames:
                memory.append(p)
            else:
                # Remove Least Recently Used page
                memory.pop(0)
                memory.append(p)

        print(f"Page {p}: {memory}")

    return page_faults

def main():
    print("\n============== LRU PAGE REPLACEMENT ==============")

    # Input pages
    pages = list(map(int, input("Enter page reference string (space-separated): ").split()))

    # Input number of frames
    frames = int(input("Enter number of frames: "))

    # Call LRU function
    faults = lru(pages, frames)

    print(f"\nTotal Page Faults (LRU): {faults}")
```

```
if __name__ == "__main__":
    main()
```

OUTPUT

```
PS C:\Users\Vismaya K\Desktop\C program> py lrureplace.py
>>

============== LRU PAGE REPLACEMENT ==============
Enter page reference string (space-separated): 7 0 1 2 0 3 0 4
Enter number of frames: 3

--- LRU Page Replacement ---
Page 7: [7]
Page 0: [7, 0]
Page 1: [7, 0, 1]
Page 2: [0, 1, 2]
Page 0: [1, 2, 0]
Page 3: [2, 0, 3]
Page 0: [2, 3, 0]
Page 4: [3, 0, 4]

Total Page Faults (LRU): 6
PS C:\Users\Vismaya K\Desktop\C program>
```

ALGORITHM

Step 1: Start

Step 2: Input the page reference string.

Step 3: Input the number of frames.

Step 4: Initialize

- an empty list memory to store pages in frames
- a counter page_faults = 0

Step 5: For each page p in the page reference string, repeat Steps 6–10.

Step 6 (Page Hit Check):

If p is already in memory:

- Remove p from its current position (because it is now most recently used)
- Insert/append p at the end of memory

Step 7 (Page Fault Case):

If p is not in memory:

- Increase page fault counter → page_faults += 1

Step 8 (Check for Free Frame):

If memory has space (i.e., len(memory) < frames):

- Add p to memory

Step 9 (No Free Frame – Replacement):

If memory is full:

- Remove the Least Recently Used page → the first element in memory
- Insert the new page at the end of memory

Step 10: Print the current memory state.

Step 11: After all pages are processed, print total page faults.

Step 12: Stop.

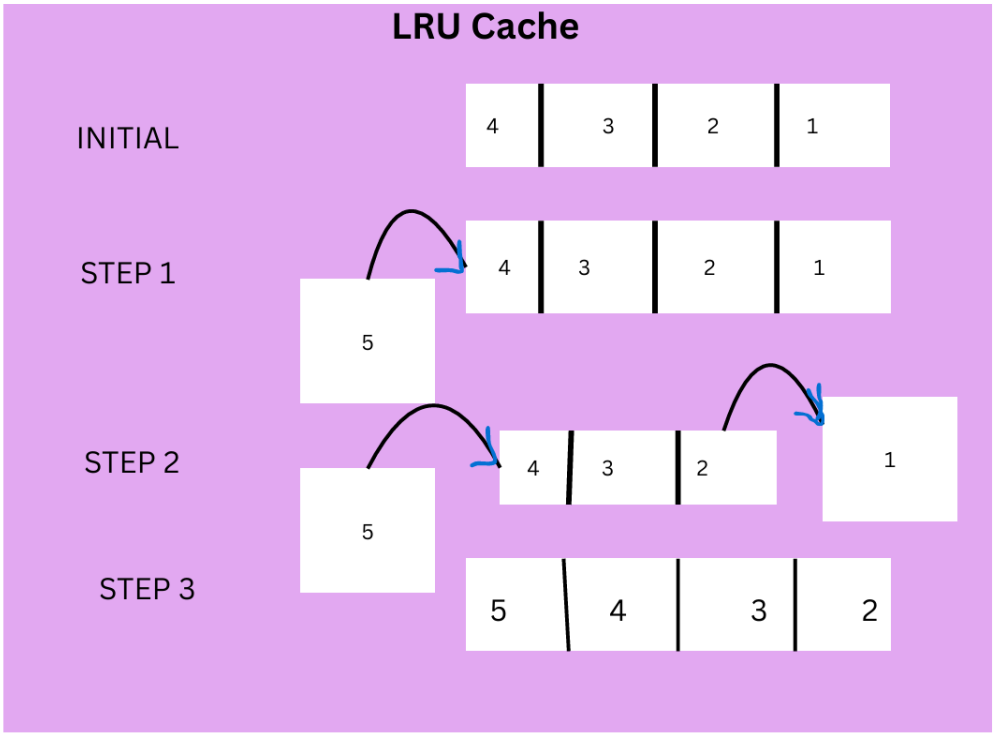
FLOW DIAGRAM EXPLANATION

The given diagram represents the general working of a Page Replacement Algorithm in an Operating System. The process begins when a program generates a memory reference and requests access to a particular page. The operating system first checks whether the requested page is already available in the main memory. If the page is present in memory, it is considered a page hit, and the process continues execution without any delay.

If the requested page is not found in memory, a page fault occurs. In this situation, the operating system checks whether there is any free space available in the memory frames. If memory is not full, the required page is directly loaded into an empty frame, and execution resumes. However, if the memory is already full, the system must decide which existing page should be removed to make space for the new one.

To select a page for removal, the operating system applies a page replacement algorithm such as First In First Out (FIFO), Least Recently Used (LRU), or Optimal Page Replacement. Based on the selected algorithm, a victim page is identified and replaced with the new page. After loading the new page into the memory frame, the process continues its execution. This cycle repeats for every memory reference until the program completes its execution. The diagram clearly illustrates how page replacement helps in efficient memory management and smooth process execution.

WORK FLOW



REPORT

This project focuses on the design and simulation of cache memory behavior by implementing page replacement techniques, which are fundamental concepts in Digital Fundamentals and Computer Architecture. Cache memory plays a crucial role in improving system performance by reducing the time required to access frequently used data. However, since cache memory has limited capacity, efficient cache management techniques such as mapping functions and replacement algorithms are required. In this simulation, emphasis is given to replacement algorithms, particularly the Least Recently Used (LRU) algorithm, to demonstrate how cache memory handles page references during program execution.

The implemented Python program simulates the LRU page replacement algorithm. The program accepts a page reference string and the number of available frames as input. It maintains a list to represent the cache memory and processes each page request sequentially. When a requested page is already present in memory, it is treated as a page hit, and the page is moved to the most recently used position. If the page is not present, a page fault occurs. If free frames are available, the page is added directly to memory. When the memory is full, the least recently used page is removed and replaced with the new page. The program displays the memory status after each page reference and calculates the total number of page faults at the end of execution.

The algorithm follows a structured approach starting from input collection, memory initialization, page hit checking, page fault handling, and replacement decision making. The flow diagram associated with this simulation visually represents the decision-making process involved in page replacement. It begins with receiving a memory reference, checks whether the page exists in memory, and proceeds accordingly based on memory availability. If memory is full, an appropriate replacement algorithm such as FIFO, LRU, or Optimal is applied to select a victim page. This systematic flow ensures efficient memory utilization and uninterrupted process execution.

Overall, this simulation provides a clear understanding of cache memory behavior and page replacement strategies. By implementing the LRU algorithm, the project demonstrates how operating systems and computer architectures manage limited memory resources effectively. The use of structured algorithms, flow diagrams, and program output analysis helps bridge theoretical concepts with practical implementation, making this project a valuable learning experience in understanding cache management and memory optimization techniques.

SCREENSHOTS OF GITLOG:

```
PS C:\Users\Vismaya K\CacheSim_Project_Team_12> git log
commit b736acd8865874f7d8ffaaef711b3591744819bd (HEAD -> VismayaKoorma-59LRU, origin/VismayaKoorma-59LRU)
Author: Vismaya K <vismayak813@gmail.com>
Date:   Sun Dec 14 10:11:57 2025 +0530

    Add LRUWORKFLOW.pdf

commit 7525be266c7537bdc5902288ecc2392b3532aad8
Author: Vismaya K <vismayak813@gmail.com>
Date:   Sun Dec 14 10:08:00 2025 +0530

    Add output.png.png image

commit 4224e652bcf6af576c344ba877d30a8eebd0a7f0
Author: Vismaya K <vismayak813@gmail.com>
Date:   Sun Dec 14 09:53:02 2025 +0530

    Add LRU page replacement Python code
```

REFLECTION

Working on the project “Simulating Cache Memory Mapping and Replacement Algorithms using Git Collaboration” was a valuable and insightful learning experience that helped me connect theoretical concepts with practical implementation. Through this assignment, I gained a deeper understanding of how cache memory functions within a computer system and why replacement algorithms are essential due to limited memory capacity. Implementing the Least Recently Used (LRU) algorithm using Python allowed me to clearly observe how page hits and page faults occur during program execution and how memory content changes dynamically based on access patterns. Writing the algorithm step by step and visualizing it through flow diagrams strengthened my logical thinking and problem-solving skills.

This project also improved my understanding of Digital Fundamentals and Computer Architecture concepts such as memory hierarchy, cache behavior, and operating system decision-making. Simulating real-time memory management helped me appreciate how efficiently modern systems optimize performance. Additionally, using Git for collaboration enhanced my practical knowledge of version control, branch management, and maintaining organized project work, which are important skills for software development. Overall, this assignment was a meaningful learning experience that enhanced both my technical and analytical skills, reinforced classroom concepts, and increased my confidence in applying theoretical knowledge to real-world computing problems.