Project Report:

Design and Implementation of a Memory Management Simulator

INDEX

1. Introduction

This project focuses on the design and implementation of a comprehensive **Memory Management Simulator**. The primary goal was to model core Operating System (OS) functions—specifically **Physical Memory Allocation**, **Multilevel Caching**, and **Virtual Memory Paging**—within a user-space C++ application.

By building this simulator, we aimed to gain a deep, hands-on understanding of:

- **Dynamic Memory Allocation:** Handling fragmentation and implementing strategies like First Fit, Best Fit, and Buddy Allocation.
- **Cache Mechanics:** Simulating associativity, replacement policies (LRU/FIFO), and write policies (Write-Back).
- **Virtualization:** Implementing page tables, address translation, and page fault handling mechanisms

2. System Architecture & Memory Layout

2.1 High-Level Architecture

The system is designed as a modular pipeline where a user request flows through distinct layers of abstraction, mimicking a real CPU memory access cycle.

Flow of Operations:

1. **CLI Interface:** Parses user commands (malloc, access, config).
2. **Virtual Memory Unit (MMU):** Translates Virtual Addresses (VA) to Physical Addresses (PA). Handles Page Faults.
3. **Cache Controller:** Intercepts Physical Addresses to check L1/L2/L3 caches. Handles Hits/Misses and Latency calculation.
4. **Physical Memory Manager:** Manages the actual raw bytes of simulated RAM.

2.2 Memory Layout & Assumptions

- **Physical Memory:** Modeled as a contiguous byte array (std::vector<char>).
    - **Addressable Unit:** Byte-addressable.
    - **Size:** Configurable at runtime (e.g., 1024 bytes to 64KB).
- **Addressing:**
    - **Virtual Address Space:** 16-bit address space (2^16 = 65,536 addresses).
    - **Physical Address Space:** Determined by the configured RAM size.
- **Assumptions:**
    - The simulation is single-threaded (single process).
    - Memory content is symbolic; we track allocation status rather than storing real user data.
    - Disk storage (Swap) is simulated symbolically by tracking "Disk Accesses" rather than writing to a file.

3. Allocation Strategy Implementations

We implemented two distinct types of memory allocators: a **List-Based Allocator** (Standard) and a **Buddy System Allocator**.

3.1 Standard Allocator (First/Best/Worst Fit)

This allocator manages memory as a linked list of **Memory Blocks**.

- **Data Structure:** std::list<MemoryBlock>
    - **Block Header:** Contains id, startAddress, size, and isFree flag .
- **Algorithms:**
    - **First Fit:** Scans the list linearly and selects the *first* free block that is large enough. Fast but may increase fragmentation at the beginning of memory.
    - **Best Fit:** Scans the entire list and selects the *smallest* free block that fits the request. Minimizes wasted space but is slower.

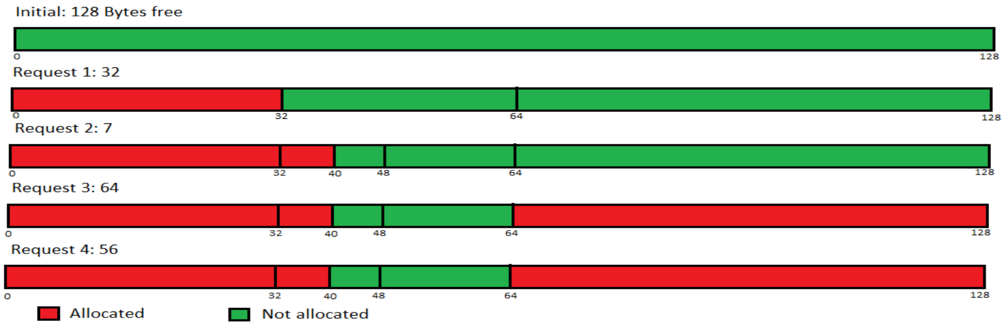
- **Worst Fit:** Selects the *largest* available free block. Intended to leave large enough chunks for future allocations, though often inefficient in practice.

- **Coalescing:**
    - **Mechanism:** Upon deallocation (free), the system immediately scans the list. If two adjacent blocks are both marked isFree, they are merged into a single larger block. This is critical for reducing **External Fragmentation**.
    - **Here Internal Fragmentation is Zero.**

3.2 Buddy System Allocator

This advanced allocator reduces external fragmentation by allocating memory in powers of two.

- **Data Structure:** An array of free lists (std::vector<std::list>), where index k stores free blocks of size 2^k.
- **Allocation Logic (Splitting):**
    1. Request size is rounded up to the nearest power of two (e.g., 30 bytes → 32 bytes).
    2. If a block of the required Order k exists, it is allocated.
    3. If not, the system searches for a block of Order k+1, splits it into two "buddies" of Order k, and repeats until the desired size is reached.
- **Deallocation Logic (Merging):**
    1. When a block is freed, its "Buddy" address is calculated using **XOR**: BuddyAddr = Addr XOR Size .
    2. If the Buddy is also free, they are merged into a block of Order k+1. This recursively bubbles up the tree.
- **Trade-off:** Very fast allocation/deallocation but suffers from **Internal Fragmentation** (wasted space inside the block due to rounding up).



4.Cache Hierarchy & Replacement Policies

The simulator implements a configurable Multilevel Cache (L1, L2, L3) system.

4.1 Cache Architecture

- Structure: Each level is explicitly modeled with Sets and Lines (Ways).

- Configurable Parameters: Size, Block Size, and Associativity are fully adjustable at runtime .
- Bit-Level Addressing:
  - Block Offset: Lower bits determining the byte within a block.
  - Set Index: (Address / BlockSize) % NumSets. Determines the specific row (Set) to check.
  - Tag: Address / (BlockSize * NumSets). Unique identifier stored in the cache line to confirm a Hit.

4.2 Write Policy: Write-Back

To accurately model modern CPUs, we implemented a Write-Back policy.

- Dirty Bit: Each cache line has a boolean dirty flag.
- Write Hit: The cache line is updated, and dirty is set to true. Main memory is not touched.
- Eviction: When a "Dirty" block is evicted (kicked out), the system simulates writing it back to Main Memory. This reduces bus traffic compared to Write-Through.

4.3 Miss Penalty Propagation (AMAT)

The simulator tracks Average Memory Access Time (AMAT).

- Latency Model:
  - L1 Access: 1 Cycle
  - L2 Access: 10 Cycles
  - L3 Access: 100 Cycles
  - RAM Access: 500 Cycles

- Logic: A miss at L1 adds the L2 latency to the total access time. A miss at L3 adds the massive RAM latency. This statistic explicitly quantifies the performance cost of cache misses.

4. Virtual Memory Model

The project simulates a paging-based Virtual Memory system, decoupling the user's view of memory from physical hardware.

5.1 Address Translation Flow

1. **Input:** User provides a Virtual Address (e.g., 0x1234).
2. **Splitting:** The address is split into **Virtual Page Number (VPN)** and **Offset**.
3. **Page Table Lookup:**
    - The system checks the **Page Table** (hash map) for the VPN.
    - **Hit:** If valid=true, the **Physical Frame Number (PFN)** is retrieved.
    - **Physical Address:** Calculated as (PFN * PageSize) + Offset.

5.2 Page Fault Handling

If the VPN is not found or valid=false, a **Page Fault** occurs:

1. **Trap:** The MMU pauses the request.
2. **Frame Allocation:** The MMU looks for a free Physical Frame.
3. **Eviction (if RAM is full):** The replacement policy selects a victim page to evict to disk (Swap). The victim's Frame is reclaimed.
4. **Load:** The new page is mapped to the Frame, and the Page Table is updated.

5.3 Replacement Policies

- **FIFO (First-In-First-Out):** Uses a queue to track page loading order. Simple but prone to Belady’s Anomaly.
- **LRU (Least Recently Used):** Uses timestamps. On every access, the page's timestamp is updated. On eviction, the page with the oldest timestamp is selected.

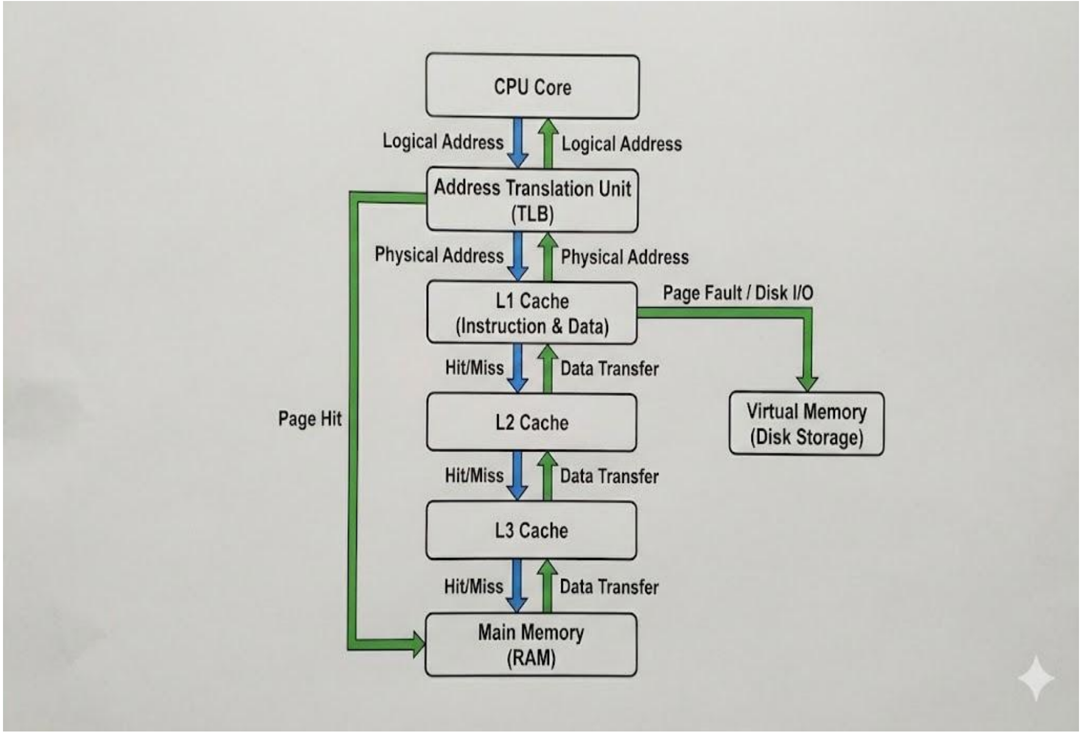
6. Integration & Order of Operations

A critical requirement was the correct integration order. The access command demonstrates this pipeline:

1. **Step 1 (MMU):** VirtualMemory :: translate(VirtualAddr)
   - Input: Virtual Address.
   - Action: Checks Page Table. Handles Page Faults (Disk I/O simulation).
   - Output: **Physical Address**.
2. **Step 2 (Cache):** CacheController :: accessMemory(PhysicalAddr)
   - Input: **Physical Address** (from Step 1).
   - Action: Checks L1 -> L2 -> L3.
   - Result: Hit or Miss (Latency calculation).

3. **Step 3 (Physical RAM):**
   - If Cache Misses all levels, data is logically fetched from the MemoryManager.

   This strictly adheres to the rule: **"Cache accesses should occur after address translation."**



7. Limitations and Simplifications

1. **Single Process:** The simulation assumes a single address space (Process ID 0). It does not handle context switching between multiple processes.

2. **Symbolic Data:** The malloc command reserves space but does not allow writing actual integers/strings into that space. The focus is on *management*, not *storage*.

3. **Unified Cache:** We simulate a unified Instruction/Data cache rather than a split Harvard architecture.

8. Conclusion

The **Memory Management Simulator** successfully meets all functional requirements. It provides a robust, interactive platform for visualizing the complex interactions between Virtual Memory, Caches, and Physical RAM. The inclusion of advanced features like **Buddy Allocation**, **Write-Back Caching**, and **AMAT Statistics** demonstrates a comprehensive understanding of Systems-Level Design.

****

Keshav Bansal

24115085