Design and Implementation of a Memory Management Simulator

-Arhat Shakya
-23116016

Abstract

This project involves the design and implementation of a comprehensive **Memory Management Simulator** in C++. The system models the complete vertical stack of computer memory, bridging the gap between Operating System software policies and Computer Architecture hardware constraints.

The simulator features a modular architecture comprising three distinct subsystems:

1. **Heap Memory Allocator:** Implements dynamic memory management using an explicit free list with support for Coalescing and multiple allocation strategies (First-Fit, Best-Fit, Worst-Fit).
2. **Virtual Memory Management Unit (MMU):** Simulates a Paging system with a single-level Page Table, handling Address Translation, Demand Paging, and LRU-based Page Replacement.
3. **Cache Memory Hierarchy:** Models a two-level (L1/L2) Set-Associative Cache system, implementing LRU eviction policies and strict Cache Coherence (Inclusion/Snooping) protocols during page faults.

The primary objective is to visualize and quantify the impact of software allocation patterns on hardware performance, specifically demonstrating **External Fragmentation**, **Locality of Reference**, and the latency penalties associated with the memory hierarchy.

High-Level Block Diagram

- **User Command (read 100) ->CLI Parser**
- **CLI ->MMU (virtual_memory)**: Request Virtual Address Translation.
- **MMU ->Page Table**: Check Validity (in_ram).
    - If Fault: **MMU ->Disk (Simulated)**: Load Page ->**RAM**.
- **MMU ->Cache Controller (cache_lvl)**: Check L1 ->L2.
    - If Miss: **Cache ->RAM**: Fetch Data.

Heap Memory Allocator (memory.hpp)

The Allocator simulates the OS kernel's dynamic memory manager (similar to malloc/free). It manages the **Virtual Address Space**, treating it as a continuous pool that can be partitioned.

Data Structure: The Doubly Linked List

The memory pool is managed as a linked list of Block nodes. This structure was chosen to allow O(1) merging of adjacent free blocks, which is impossible with simple bitmaps.

- **struct Block**:
  - **size**: Capacity of the chunk.
  - **address**: The logical starting address (Virtual Address).
  - **block_id**: A unique integer ID assigned upon allocation. This allows free_block(id) to find specific allocations without knowing the address.
  - **is_free**: Boolean flag. true = Hole (Free), false = Process Data (Allocated).
  - **prev / next**: Pointers to neighbors. These are critical for the **Coalescing** logic.
- **struct Memory**:
  - **id_counter**: A global counter that increments on every malloc, ensuring every block gets a unique ID.
  - **allocator**: A state variable (1, 2, or 3) determining which strategy (first, best, worst) is currently active.
  - int TOTAL_MEMORY: The total size of the simulated heap.

Allocation Algorithms

The simulator implements three standard strategies to solve the **Dynamic Storage-Allocation Problem**, each addressing **External Fragmentation** differently.

A. First-Fit (malloc_first)

- **Algorithm:** Linearly scans the list from the head. Stops at the first free block where block->size >= requested_size.
- **Splitting:** If the found block is larger than requested, it is split into two:
  1. **Allocated Block:** Size = requested_size.
  2. **New Hole:** Size = original_size - requested_size.
- **Analysis:**
  - **Time Complexity:** O(N) worst-case, but often faster as it stops early.
  - **fragmentation:** Tends to accumulate small "splinters" at the beginning of the list.

B. Best-Fit (malloc_best)

- **Algorithm:** Traverses the entire list. It tracks the free block that minimizes the value of (block->size - requested_size).
- **Analysis:**
  - **Time Complexity:** Strict O(N) (must scan all nodes).
  - **Fragmentation:** Reduces "wasted space" (Internal Fragmentation) but creates tiny, unusable holes that are hard to fill later.

C. Worst-Fit (malloc_worst)

- **Algorithm:** Traverses the entire list to find the free block with the **maximum** size.
- **Analysis:**
    - **Philosophy:** By splitting the largest block, the remaining hole will be as large as possible, theoretically making it more useful for future allocations.
    - **Drawback:** Destroys large contiguous regions, making it impossible to service large requests later.

Deallocation Mechanism (free_block)

This function implements **Immediate Coalescing**.

1. **Marking:** The target block is flagged as is_free = true.
2. **Backward Merge:** If block->prev exists and is free, the current block is merged into the previous one. The current node is deleted.
3. **Forward Merge:** If block->next exists and is free, the next node is merged into the current one. The next node is deleted.

- **Impact:** This logic guarantees that no two free blocks ever exist side-by-side, maintaining the largest possible contiguous free regions.

Cache Memory System (cache.hpp)

This module simulates hardware caches (L1 and L2). It uses a **Set-Associative Mapping** technique, which is the standard design for modern CPUs (balancing the speed of Direct Mapped and flexibility of Fully Associative caches).

Data Structures

- **struct cacheline:** Represents a single slot in the cache.
    - int tag: The upper bits of the address, used to identify if this is the correct block.
    - int data: Placeholder for content.
- **struct cache_lvl:** Represents a cache level (L1 or L2).
    - vector<deque<cacheline>> cache: The storage. It is a Vector of Sets.
        - Each "Set" is a **Deque (Double-ended queue)**.
        - Using a Deque allows easy implementation of the **LRU (Least Recently Used)** policy: The most recently used item is moved to the front, and eviction happens at the back.
    - int set_bits, offset_bits: Pre-calculated masks used to decode addresses bitwise.

Core Functions

1. **Address Decoding:**
    - The system splits a Physical Address into three parts:
        - **Offset:** Lower bits determining the specific byte inside the block (e.g., bits 0-4 for 32B blocks).
        - **Set Index:** Middle bits determining which "row" (Set) to check.
        - **Tag:** Upper bits used to verify if the data matches the request.
2. **cache_read(int phy_address):**
    - **Logic:** Calculates Tag and Set Index. It searches that specific Set's deque for the Tag.
    - **Hit:** If found, it moves that cache line to the front of the deque (LRU update) and returns true.
    - **Miss:** Returns false.
3. **cache_insert(int phy_address):**
    - **Logic:** Inserts a new tag at the front of the correct Set.
    - **Eviction Policy:** If the Set is full (size == associativity), it removes the element at the back of the deque (the LRU victim) before pushing the new one to the front.
4. **cache_flush(int phy_address):**
    - **Purpose:** Ensures **Cache Coherence**.
    - **Logic:** Forces the removal of a specific address range from the cache. This is called by the VM system when a page is evicted from RAM. Without this, the cache would hold "stale" data for a physical address that has been reassigned to a different virtual page.
5. **clear_cache:**
   Clears the deques in L1/L2, simulating a cache flush.

Virtual Memory Unit (virtual_memory.hpp)

This module simulates the **Memory Management Unit (MMU)** and the OS Paging Kernel. It introduces the concept of **Paging**, decoupling the user's view of memory (Virtual) from the actual hardware storage (Physical RAM).

Architectural Parameters

- **Page Size:** Defines the granularity of memory management (e.g., 256 bytes).
- **Virtual Space:** The total addressable memory (User perception).
- **Physical RAM:** The actual limited hardware memory (e.g., 4096 bytes).

Data Structures

- **PageTableEntry:**
  - **in_ram**: Valid bit. 1 = In Memory, 0 = On Disk.
  - **frame_number**: The Physical Frame Number (PFN) where the page resides.
- **Frame**: Represents a physical RAM slot. Contains page_id to track reverse mapping (Frame to Page).
- **LRU (Deque of Integers)**: This is the core of the Page Replacement Policy.
  - **Logic:** It stores Frame IDs. The **Front** represents the Most Recently Used frame. The **Back** represents the Least Recently Used (Victim).
- **empty_frames (Deque)**: A stack of currently unused RAM frames.

Core Functions

ram_read(int address)

- **OS Concept: Address Translation & Page Hits**.
- **Logic:**
  1. **VPN Calculation:** Computes Virtual Page Number = address / PageSize.
  2. **Page Table Lookup:** Checks PageTable[VPN].in_ram.
  3. **Hit:** If the flag is true:
     - The page is in RAM.
     - **LRU Update:** It finds the frame number in the LRU deque, removes it, and pushes it to the front. This signals "This frame was just used, don't evict it soon."
     - Returns true.
  4. **Miss:** If the flag is false, returns false, signaling a Page Fault.

ram_insert(int address)

- **OS Concept: Page Fault Handling & Demand Paging**.
- **Logic:**
  1. **Check Free Space:** Looks at empty_frames.
  2. **Eviction (Swap Out):** If RAM is full (empty_frames is empty):
     - **Select Victim:** Picks the frame ID at the **back** of the LRU deque.
     - **Flush Caches (Critical):** Before reusing this frame, it calculates the physical memory range of this frame (e.g., Address 4096 to 8191) and calls cache_flush for every block in that range. This enforces coherence.
     - **Unmap:** Updates the old page's entry to in_ram = false.
  3. **Placement (Swap In):**
     - Takes a free frame (either from the free list or the one just evicted).
     - **Map:** Updates the new page's entry to point to this frame and sets in_ram = true.

        - **Update LRU:** Pushes this frame ID to the front of the LRU deque.

total_read(int address)

- **OS Concept: The Memory Hierarchy Pipeline**.
- **Logic:** This function orchestrates the entire flow.
    1. **TLB/MMU Check:** Calls ram_read(address).
        - If it returns false, the hardware traps to the OS, which calls ram_insert(address) (handling the Page Fault).
    2. **Physical Address Generation:** Once the page is guaranteed to be in RAM, it computes the Physical Address:
        - PA = (FrameNumber * PageSize) + Offset.
    3. **L1 Cache Access:** Attempts level1->cache_read(PA).
        - If **Hit**: Returns data immediately (Fastest).
    4. **L2 Cache Access:** If L1 missed, attempts level2->cache_read(PA).
        - If **Hit**: Simulates copying data from L2 to L1 (level1->cache_insert), then returns.
    5. **RAM Access:** If both caches miss:
        - It "fetches" from RAM.
        - **Backfilling:** It inserts the data into L2, and then into L1. This behavior simulates the **Inclusive Property** (or at least the data flow) of modern hierarchies.

clear_ram()

- **Purpose:** Simulates an OS Reboot.
- **Mechanism:** Resets all Page Table valid bits to 0, marks all RAM frames as free, clears the LRU queue, and triggers cache clears.

Conclusion

The hierarchical memory simulator successfully demonstrates the complex interplay of modern computer architecture components.

- **Heap Allocation** experiments reveal that **Best-Fit** minimizes internal fragmentation at the cost of processing time, while **Coalescing** is essential for long-term system stability.
- **Virtual Memory** simulation validates the effectiveness of **LRU Paging** in allowing large virtual address spaces to exist on limited physical hardware, albeit with the performance penalty of **Page Faults** (Thrashing) when the working set exceeds RAM size.
- **Cache Simulation** highlights the critical role of **Locality of Reference**. Sequential access patterns yield high hit rates, while random access patterns (simulated via CLI scripts) degrade performance significantly, visualizing the "Memory Wall" problem in modern computing.