Design and Implementation of a Memory Management Simulator

-Arpit Bhargava
-23113032

---

1. Memory Layout and System Assumptions

The simulator assumes physical memory as collection of memory blocks implemented using **Doubly linked list**. The system is divided into three distinct layers that communicate through a structured interface:

- **Virtual Memory Space:** A large, logical address space (e.g., 32KB) exposed to "processes."
- **Physical RAM:** A smaller, fixed-size hardware memory (e.g., 8KB) partitioned into **Frames**.
- **Cache Hierarchy:** High-speed storage (L1 and L2) that sits between the CPU and Physical RAM.

Memory representation :-

Units -> contiguous block of integers.

Block structure-> memory is divided into nodes, where each node represents a chunk of memory.

  - startaddress ;- integer offset where block begins
  - Size -> size of allotted memory
  - Id -> unique integer of each block
  - Flag -> free or allotted representation
  - next / prev -> next and previous block pointers

Assumption: The Page Size, Cache Block Size, and Physical RAM size are all powers of two to allow for efficient bitwise operations during address translation.

---

2. Allocation Strategy Implementations

The **Physical Memory Allocator** (memory.hpp) manages the heap using a **Doubly Linked List** of Block structures. Each block represents either a "Hole" (free) or "Allocated" space.

Strategies:

- **First-Fit:** Scans the list from the head and selects the first available hole large enough to satisfy the request. It is efficient in speed but can lead to "shards" at the beginning of memory.
- **Best-Fit:** Scans the entire list to find the hole that is closest in size to the request. This minimizes wasted space (internal fragmentation) but leaves behind tiny, unusable holes.
- **Worst-Fit:** Scans for the largest available hole. The philosophy is that splitting a large hole will leave a remaining hole large enough to be useful for future allocations.

Deallocation & Coalescing ->

- Freeing: When Free(id) is called, the simulator scans for the block with the matching ID and sets its Hole flag to true.
- Coalescing: Immediately after freeing, the allocator checks the NextBlock and PrevBlock. If adjacent blocks are also holes, they are merged into a single larger block to reduce external fragmentation.

---

3. Cache Hierarchy and Replacement Policy

The system implements a **Set-Associative Cache** (cache.hpp).

- **Hierarchy:** Supports L1 and L2 levels. If a request misses L1, it probes L2; if it misses both, it fetches from Physical RAM and populates both cache levels (Inclusive Policy).
- **Set Calculation:** Uses bit-masking to extract Tag, Index (Set), and Offset.
- **Replacement Policy:** Implements **LRU (Least Recently Used)** via std::deque. When a set is full, the block at the back() of the deque (the oldest) is evicted, and the new block is push_front().

---

4. Virtual Memory Model and Translation

The Virtual Memory module (virtual.hpp) bridges the gap between logical process addresses and physical RAM.

- **Paging:** The address space is divided into **Pages** (Virtual) and **Frames** (Physical).
- **Page Table:** A structure of PageTableEntry tracks which virtual pages are currently mapped to physical frames and their validity.
- **Demand Paging:** Pages are only loaded into RAM when accessed. If a page is not in RAM, a **Page Fault** is triggered.

---

5. Address Translation Flow

Every "read" command follows this rigorous hardware-simulated path:

1. **Virtual to Physical:** The Virtual Address (VA) is split into a Virtual Page Number (VPN) and Offset.
2. **Page Table Lookup:**
    - **Hit:** VPN is found in the table; the Frame Number is retrieved.
    - **Fault:** If the page is missing, the **LRU Replacement Policy** evicts a frame from RAM to make room for the new page.
3. **Cache Probing:** The resulting Physical Address (PA) is used to check L1, then L2.
4. **Final Fetch:** If a Cache Miss occurs at all levels, the system accesses "Physical RAM."

---

6. Limitations and Simplifications

- **Single Process:** The current model simulates a single-process environment (one Page Table).
- **Write Policy:** The simulator primarily focuses on "Read" operations; a "Write-Back" or "Write-Through" consistency model is not fully implemented for the cache.
- **No Disk I/O:** Page faults assume an instantaneous "swap-in" from disk, as the disk latency is not simulated in the timing.
- **Buddy System:** While the allocation strategy supports various fits, the specific power-of-two "Buddy System" (which splits blocks into exactly halves) is not utilized

in the current linked-list implementation; instead, it uses a flexible splitting method to minimize external fragmentation.