Memory Management Simulator Design Document

January 8, 2026

Contents

1 Introduction

This document describes the design and implementation of the Memory Management Simulator. The system models a complete memory hierarchy, including a physical memory manager with multiple allocation strategies, a Buddy System allocator, a configurable three-level cache hierarchy, and a virtual memory system with paging and disk latency simulation.

2 Memory Layout and Architecture

2.1 Physical Memory Model

The physical memory is simulated as a contiguous block of bytes, implemented using a `std::vector<char>`. This vector represents the raw storage (RAM) available to the system. The total size is configurable at initialization via the CLI.

2.2 Block Structure

Memory allocations are tracked using a `BlockHeader` structure embedded within the physical memory. Each block contains:

- **Size**: The usable payload size.
- **Padding**: Internal fragmentation bytes to ensure alignment.
- **Status**: A boolean flag (`is_free`) indicating allocation status.
- **ID**: A unique integer identifier for tracking allocations.
- **Pointers**: `next` and `prev` pointers to maintain a doubly linked list for traversal.

All allocations are aligned to 8-byte boundaries to simulate architectural alignment requirements.

3 Allocation Strategies

The simulator implements four dynamic memory allocation strategies within the `MemoryManager` class.

3.1 Standard Allocators

- **First Fit**: Scans the free list from the head and selects the first block capable of satisfying the request. Fast but prone to fragmentation at the start of memory.
- **Best Fit**: Scans the entire list to find the block closest in size to the request. Minimizes internal fragmentation but incurs higher search overhead.
- **Worst Fit**: Scans the entire list to find the largest available block. Intended to leave large usable chunks after splitting.

4 Buddy System Design

The `BuddyAllocator` is a distinct module that manages memory in power-of-two blocks.

4.1 Design

- **Free Lists**: An array of linked lists, where index i stores blocks of order i (size 2^i).
- **Splitting**: Requests are rounded up to the nearest power of two. If a block of the required order is unavailable, a larger block is recursively split into two "buddies".
- **Coalescing via XOR**: When a block of order k is freed, the allocator calculates the address of its buddy using the bitwise Exclusive-OR (XOR) operation:

  $$BuddyAddr = BlockAddr \oplus Size$$

  This mathematical property ensures that every block has a unique, deterministic buddy. If the buddy is also free, they are merged into a single block of order $k + 1$, and the process repeats recursively.

4.2 Coalescing Example

To illustrate the XOR logic:

1. Assume a block at address `0x100` with size `0x100` (256 bytes) is freed.
2. The potential buddy address is calculated as: `0x100` $\oplus$ `0x100` = `0x000`.
3. If the block at `0x000` is also free and has the same size (256 bytes), they merge.
4. The new merged block starts at the lower address (`0x000`) and has a size of 512 bytes (`0x200`).

5 Cache Hierarchy

The simulator models a sophisticated **three-level** cache hierarchy.

5.1 Configuration

The cache levels are initialized with specific parameters:

- **L1 Cache**: 64 Bytes, 1-way (Direct Mapped), 8 Byte blocks.
- **L2 Cache**: 256 Bytes, 2-way Set Associative, 8 Byte blocks.
- **L3 Cache**: 1024 Bytes, 8-way Set Associative, 64 Byte blocks.

5.2 Replacement Policy Implementations

The system supports multiple replacement policies, configurable at runtime:

- **FIFO (First-In First-Out)**: Evicts the cache line that was inserted earliest. The system tracks the insertion order to identify the oldest resident block, regardless of how recently it was accessed.
- **LRU (Least Recently Used)**: Evicts the cache line that has not been accessed for the longest duration. Each line maintains a `last_access_time` timestamp, which is updated on every hit. The entry with the oldest timestamp is selected for eviction.
- **LFU (Least Frequently Used)**: Evicts the cache line with the fewest total accesses. Each block maintains an `access_count`. On eviction, the block with the lowest count is removed. Ties are broken using the LRU policy.

6 Virtual Memory System

The `VirtualMemoryManager` simulates paging, address translation, and page faults.

6.1 Address Translation Flow

When the CPU requests a virtual address, the system follows a fixed translation sequence:

1. **Bit Slicing**: The Virtual Address is split into a *Virtual Page Number (VPN)* and an *Offset*.
$$Offset = VA \ \& \ (PageSize - 1)$$
$$VPN = VA \gg \log_2(PageSize)$$
2. **Page Table Lookup**: The system consults the Page Table using the VPN.
   - If the **Valid Bit** is set, the frame number is retrieved.
   - If the Valid Bit is clear, a **Page Fault** is triggered (loading data into a free frame or evicting a victim).
3. **Physical Address Generation**: Once the Physical Frame Number (PFN) is resolved, the physical address is computed:
$$PhysicalAddress = (PFN \times PageSize) + Offset$$
4. **Cache Access**: The resulting Physical Address is sent to the L1 Cache to initiate the memory access pipeline.

6.2 Example Translation

Consider a system with a **64-byte Page Size** and a request for Virtual Address **100**:

- **Offset**: $100 \pmod{64} = 36$.
- **VPN**: $100/64 = 1$.
- The system checks the Page Table for Page 1. If mapped to Frame 5, the Physical Address becomes $(5 \times 64) + 36 = 356$.

6.3 Page Replacement Policies

- **FIFO**: Uses a queue to track page load order.
- **LRU**: Uses access timestamps to identify the least recently used page.
- **CLOCK**: Implements the "Second Chance" algorithm using reference bits.

7 System Interface (CLI)

The simulator is driven by a command-line interface. Below is an example of the statistics output generated by the system after a series of memory access operations, illustrating the detailed tracking of cache hits, misses, and page faults.

```
> stats

=== Memory System Statistics ===
Memory Utilization: 0% (0/4096 bytes)
Internal Fragmentation: 0 bytes
External Fragmentation: 0%
Allocation Requests: 0
Successful Allocs:   0
Success Rate:        0%
==============================

=== Cache Statistics ===
L1 Cache Stats:
  Hits: 0
  Misses: 12
  Hit Rate: 0.00%
L2 Cache Stats:
  Hits: 1
  Misses: 11
  Hit Rate: 8.33%
L3 Cache Stats:
  Hits: 2
  Misses: 9
  Hit Rate: 18.18%
========================

=== Virtual Memory Statistics ===
  Page Faults: 9
  Page Hits:   3
  Hit Rate:    25.00%
================================
```

8 Verification and Testing

This section demonstrates the system's correctness using specific test scenarios.

8.1 Test Case 1: Cache Hierarchy & Conflict Misses

This scenario tests the simulator's handling of cache associativity and conflict misses using a specific stride pattern.

Test Input:

```
init 4096
set cache policy fifo
enable_vm 64
# Access patterns with Stride 64
read 0, 64, 128, 192, 256, 320, 384, 448
# L1 is full (8 blocks). Next read causes eviction in Set 0.
read 512
# Access 0 again. Should miss in L1/L2 but hit L3.
read 0
stats
```

Results Analysis: The simulator correctly reports 10 L1 Misses (9 initial + 1 re-access) and 1 L3 Hit. The stride of 64 bytes forces all addresses into Set 0 of the Direct-Mapped L1, causing immediate eviction. The L3 (8-way) retains the data, proving inclusive hierarchy behavior.

8.2 Test Case 2: LRU Policy and Thrashing

This scenario tests the **LRU** replacement policy and the hierarchy depth during "thrashing."

Test Input:

```
init 4096
set cache policy lru
enable_vm 64
# Access 0..448 (Stride 64) -> Fills L1 Set 0
read 0
# Read 512 -> Evicts 0 from L1/L2
read 512
# Check retention
read 0
read 64
stats
```

Results Analysis:

- **L1 Hit Rate 0%**: Confirms constant conflict misses in Set 0.
- **L3 Hits**: The final accesses to address 0 and 64 are found in L3.
- **Conclusion**: The hierarchy successfully preserves data in higher levels (L3) despite thrashing in lower levels (L1), validating the set-associative design and replacement logic.

9 Limitations

- **Single Threaded**: The simulator does not model race conditions.
- **No TLB**: Address translation always accesses the Page Table directly (TLB logic is conceptual).
- **Data Persistence**: The system focuses on tracking the *logic* of access (hits/misses) rather than persisting user data values through all eviction layers.