# Memory Management Simulator: Full Technical Design

System Architecture Report

January 8, 2026

## Contents

# 1 Physical Memory Allocator: Hybrid Data Structure Design

## 1.1 Overview

The Physical Memory Allocator manages a contiguous block of memory using a hybrid approach—combining a doubly linked list, a balanced binary search tree, and a hash map—to provide efficient allocation, fast block lookup, and immediate coalescing of adjacent free segments.

## 1.2 Core Data Structures

### 1.2.1 A. The Physical Sequence (`std::list<block>`)

This list represents the physical layout of memory. Each node is a block representing a segment of memory, whether free or occupied.

```
struct block {
    std::size_t addr;     // Starting byte address
    std::size_t len;      // Size of the segment in bytes
    bool is_free;         // Status: true if available for allocation
    int id;               // Unique identifier ( -1 if free )
};
```

**Role:** Maintains spatial adjacency. Necessary for coalescing (merging) neighboring free blocks.
**Time Complexity:** $O(1)$ for inserting/deleting nodes once the iterator is known.

### 1.2.2 B. The Free Block Index (`std::set<iterator, Compare>`)

A balanced Binary Search Tree (BST) that stores iterators to all blocks where `is_free == true`.

```
struct Compare {
    using is_transparent = void; // Enables searching by size_t directly

    bool operator()(const list_it& a, const list_it& b) const {
        if (a->len != b->len) return a->len < b->len; // Primary: Size
        return a->addr < b->addr;                     // Secondary: Address
    }
};
```

**Role:** Enables fast searching for available blocks based on size.
**Key Feature:** The `is_transparent` tag allows `index.lower_bound(requested_size)` to work without creating a dummy iterator object.

### 1.2.3 C. The ID Registry (`std::unordered_map<int, iterator>`)

**Role:** Allows the `free(id)` operation to find the target block instantly without traversing the entire list in $O(1)$ average time.

## 1.3 Allocation Policies

### 1.3.1 First Fit

**Logic:** Scans the physical list (`mem_list`) from the beginning and selects the first block that is large enough.
**Complexity:** $O(M)$, where $M$ is the total number of blocks (free + occupied).

#### 1.3.2 Best Fit

**Logic:** Uses `index.lower_bound(size)` on the BST index.
**Complexity:** $O(\log N)$, where $N$ is the number of free blocks.

#### 1.3.3 Worst Fit

**Logic:** Accesses the last element of the set via `*index.rbegin()`.
**Complexity:** $O(\log N)$.

### 1.4 Core Helper Functions

- **`addblock(iterator it, size_t size)`**: Marks a block as occupied, assigns an ID, and updates the `id_map`. If the block is larger than requested, it splits the block and inserts the remainder back into `mem_list` as a new free block.
- **`add_index` / `del_index`**: Manages the `std::set`. Updates trigger $O(\log N)$ tree reshuffling.

### 1.5 Coalescing Logic (The `free` Operation)

When a block is released via `free(id)`, the allocator merges the newly freed block with physical neighbors:

1. **Lookup:** Find the block in $O(1)$ using the `id_map`.
2. **Mark Free:** Set `is_free = true` and `id = -1`.
3. **Check Neighbors**: Check `std::prev` and `std::next`. If a neighbor is free, merge it.
4. **Update Index**: Remove old neighbor iterators and add the new merged block iterator.

# 2 Hierarchical N-Way Set-Associative Cache Design

## 2.1 System Overview

This system implements a multi-level cache hierarchy (L1 and L2) to bridge the latency gap between the CPU and RAM. It utilizes a Set-Associative strategy, which organizes memory into sets, each containing multiple "ways" (lines).

## 2.2 Core Data Structures

### 2.2.1 A. The Storage Hierarchy

The cache memory is structured as a nested collection of vectors to simulate physical hardware banking.

- **Cache Line (`struct CacheLine`)**: The smallest unit. Stores the `tag`, `valid` bit, `modified` (dirty) bit, and the physical address.
- **Cache Set (`struct CacheSet`)**: A collection of $N$ Cache Lines (where $N$ is the associativity).
- **The Cache Body (`std::vector<CacheSet> sets`)**: The primary memory structure, indexed via address bits.

### 2.2.2 B. Address Decoding

A 64-bit physical address is parsed into:

- **Tag**: Higher bits used for comparison to verify a hit.
- **Index**: Middle bits used to select the specific `CacheSet`.
- **Offset**: Lower bits identifying the location within the block.

## 2.3 Initialization & Policy Management

### 2.3.1 A. `initFIFO()` & `initLRU()`

- `initFIFO()`: Initializes `fifoNextVictim` pointers to 0 for each set. ($O(S)$ complexity).
- `initLRU()`: Populates the `lruLists` with initial indices and stores their positions in `lruMaps`. ($O(S \times N)$ complexity).

### 2.3.2 B. `updatePolicyOnHit(setIndex, lineIndex)`

This function is called whenever a cache hit occurs to update the replacement metadata.

- **LRU Logic**: Uses `list.splice` to move the accessed `lineIndex` from its current position in the list to the front (MRU position). Because the position is retrieved from `lruMaps`, this is an $O(1)$ operation.
- **FIFO Logic**: Does nothing. FIFO only updates metadata during replacement, as hits do not change the "arrival order" of the block.

#### 2.3.3 C. `updatePolicyOnReplace(setIndex, lineIndex)`

This function updates the metadata after a new block is brought into the cache.

- **LRU Logic**: Calls `updatePolicyOnHit`. The newly loaded block becomes the "Most Recently Used."
- **FIFO Logic**: Increments the `fifoNextVictim[setIndex]` pointer using modulo arithmetic: `(ptr + 1) % associativity`. This ensures a round-robin eviction pattern.

#### 2.3.4 D. `getVictimIndex(setIndex)`

When a cache miss occurs and a set is full, this function identifies which line index should be evicted.

- **LRU Logic**: Returns the index at the `back()` of `lruLists[setIndex]`. Since the system maintains MRU at the front, the back always contains the least recently accessed way.
- **FIFO Logic**: Returns the current value of `fifoNextVictim[setIndex]`, identifying the oldest block according to the round-robin schedule.
- **Complexity**: This is an $O(1)$ operation for both policies.

### 2.4 The Access Algorithm (`access`)

The `access(address, write)` function is the primary entry point for memory requests.

- **Decoding**: The address is split into Tag, Index, and Offset.
- **Direct-Mapped Shortcut**: If associativity is 1, it checks `lines[0]` immediately without a loop.
- **Set Traversal ($O(N)$)**: The system iterates through the lines of the target set.
  - **On Hit**: If a valid line matches the tag, hits are incremented. If it's a write, the `modified` bit is set. `updatePolicyOnHit` is called to refresh the priority.
- **Miss Handling**: If no tag matches, the system first looks for an invalid (empty) line.
- **Eviction**: If the set is full, `getVictimIndex` is called. For LRU, it returns the `back()` of the list; for FIFO, it returns the current `fifoNextVictim` pointer.
- **Final Update**: The victim line is populated with the new tag and address, and `updatePolicyOnReplace` is called to finalize the metadata state.

### 2.5 Cache Hierarchy & Performance Stats

The `cacheHierarchy` class coordinates L1 and L2. If L1 misses, it automatically triggers an access to L2.

#### 2.5.1 A. Core Metrics

**Hit Rate**: $\frac{\text{Hits}}{\text{Hits}+\text{Misses}}$

#### 2.5.2 B. Miss Penalty

**L1 Miss Penalty**: $T_{L2\_Hit} + (\text{L2\_Local\_Miss\_Rate} \times T_{RAM\_Access})$

### 2.5.3 C. Average Access Time (AAT)

**AAT**: $T_{L1\_Hit} + (\text{L1\_Miss\_Rate} \times \text{L1\_Miss\_Penalty})$

# 3 Virtual & Physical Memory Interaction Design

## 3.1 Overview

This module simulates the operating system's memory management unit (MMU) and paging subsystem. It manages the mapping between a process's Virtual Address Space and the system's Physical RAM Frames, handling address translation, page faults, and page replacement.

## 3.2 Core Data Structures

### 3.2.1 A. Process View (The Virtual Memory)

Each process has its own `VirtualMemory` instance containing its private mapping.

- **Page Table** (`std::unordered_map<uint64_t, PageTableEntry>`): Maps a Virtual Page Number (VPN) to a Physical Frame.
- **Page Table Entry** (`struct PageTableEntry`):
  - `valid`: Boolean flag indicating if the page is currently loaded in RAM.
  - `frameNumber`: The specific physical frame index assigned to this page.

### 3.2.2 B. Hardware View (The Physical Memory)

The `PhysicalMemory` class represents the system's RAM, shared by all processes.

- **Frame Registry** (`std::vector<FrameInfo> frames`): An array representing physical RAM slots.
  - `isUsed`: Indicates if the frame is occupied.
  - `ownerPID` / `ownerVPN`: Back-pointers used to notify a process when its page is evicted.
- **Process Registry** (`std::unordered_map<int, VirtualMemory*>`): A "Bridge" map that allows Physical Memory to look up a process by its PID to invalidate pages during eviction.

## 3.3 The Address Translation Pipeline (`translate`)

The `translate(virtualAddr)` function is the primary entry point for memory access.

- **Decoding**: The virtual address is split into a VPN (Virtual Page Number) and an Offset based on the page size.
- **Page Table Lookup**:
  - **Page Hit**: If `pageTable[vpn].valid == true`, the system retrieves the `frameNumber`, updates the replacement metadata (if LRU), and returns the physical address.
  - **Page Fault**: If the entry is invalid, the system increments the `pageFaults` counter and calls `physMem->allocate()`.
- **Frame Assignment**: The new frame returned by the allocator is stored in the page table, marked as valid, and the translated address is calculated: `(frame << offsetBits) | offset`.

## 3.4 Allocation and Eviction Logic (`allocate`)

When a process needs a frame but RAM is full, the allocator must select a "victim" to evict.

### 3.4.1 A. Initial Allocation (The "IsFull" phase)

Before RAM is full, the system simply assigns the `nextFreeFrameIndex`. This is a constant time $O(1)$ operation.

### 3.4.2 B. Page Replacement Policies

- **FIFO (First-In, First-Out)**: Uses a `fifoPointer` to treat frames as a circular buffer. The oldest loaded frame is evicted first.
- **LRU (Least Recently Used)**: Uses a `std::list` (order) and `std::unordered_map` (iterator lookup). On every access, the frame is moved to the front. The `back()` of the list is the victim.

### 3.4.3 C. The Invalidation Bridge

When a frame is selected for eviction:

- **Retrieve** the `victimPID` and `victimVPN` from the frame's metadata.
- **Use** the `processRegistry` to find the `VirtualMemory` object belonging to that PID.
- **Call** `invalidatePage(vpn)`: The process marks that specific VPN as `valid = false`. This ensures consistency between the Page Table and actual RAM state.

# 4 System Integration & CLI

## 4.1 Overview

The Simulator CLI acts as a functional wrapper around the core memory modules. It translates user-entered commands into direct method calls on class instances, providing an interactive way to observe state changes in the Allocator, Cache, and Virtual Memory systems.

## 4.2 CLI Functional Mapping

### 4.2.1 A. Allocator Mode

This mode provides a front-end to the `Memory` class.

- **`init <size>`**: Triggers the constructor `std::make_unique<Memory>(size)`. This initializes the `mem_list` with a single large free block and populates the index (BST).
- **`malloc <size>`**: Calls either `allocate_bestfit`, `allocate_worstfit`, or `allocate_firstfit` based on the selected strategy. It returns the unique ID generated by the allocator.
- **`free <id>`**: Executes `mem->free(id)`, which triggers the hash map lookup and the neighbor-coalescing logic.
- **`stats / dump`**: Accesses the internal metrics (Success Rate, External Fragmentation) and prints the current state of the `mem_list`.

### 4.2.2 B. Cache Simulator Mode

This mode orchestrates the `Cache` and `cacheHierarchy` classes.

- **`init`**: Instantiates two `Cache` objects (L1 and L2) and passes their pointers to a new `cacheHierarchy` instance. This sets up the internal sets vectors and initializes replacement metadata (FIFO/LRU).
- **`access <addr> <type>`**: Calls `hierarchy->access(physicalAddress, isWrite)`. This initiates the L1 tag search, falling back to L2 on a miss.
- **`stats`**: Triggers the calculation of Average Access Time (AAT) and Miss Penalties based on the recorded hits and misses.

### 4.2.3 C. Virtual Memory Mode

This mode interfaces with `PhysicalMemory` and multiple `VirtualMemory` instances.

- **`init`**: Creates the `PhysicalMemory` object, defining the global RAM frame count and page size.
- **`access <pid> <addr>`**: Checks if a `VirtualMemory` object exists for that PID; if not, it creates one. It then calls `translate(virtualAddr)`, which potentially triggers a page fault and frame allocation.
- **`status`**: Calls `pm->printStatus()` to show which PID and VPN currently occupy each physical frame in RAM.

### 4.3 Integrated Mode (The Full Stack)

The Integrated mode demonstrates the end-to-end data flow from a software request to hardware execution.

#### Initialization

Simultaneously initializes a `PhysicalMemory` pool and a multi-level `Cache` hierarchy.

#### Access Flow

1. User provides a **Virtual Address**.
2. The CLI calls `VirtualMemory::translate`, which converts it to a **Physical Address** (handling any page faults or evictions internally).
3. The resulting **Physical Address** is immediately passed to `cacheHierarchy::access`.
4. The system reports the final hardware result (Cache Hit/Miss).

# 5 Project Assumptions & Implementation Limitations

## 5.1 Core Technical Assumptions

- **Data Structures**: The Page Table uses `std::unordered_map` for efficiency.
- **Memory Model**: The simulation does not account for the memory required to maintain virtual memory structures or store page tables within physical memory frames.
- **Sizing**: All memory components (RAM, Pages, Cache Blocks) are assumed to be powers of two.
- **Process Handling**: PIDs are assumed unique, and the correct ID is provided when freeing memory.

## 5.2 Consolidated Limitations

- **Memory Protection**: Lack of hardware-level access controls, user/kernel mode distinction, or protection rings.
- **Address Space**: Utilizes a linear addressing model only; no support for complex segmentation or disk-backed demand paging.
- **Fragmentation & Allocation**: Only adjacent block coalescing is supported (no compaction). Allocation is limited to a fixed pool size defined at construction.
- **Cache & Policies**: No cache coherency protocols, prefetching mechanisms, or multi-core considerations. Only basic LRU and FIFO policies are implemented.
- **Translation**: Uses a single-level page table without a Translation Lookaside Buffer (TLB) or multi-level table support.
- **Simulation Execution**: Strictly single-threaded simulation; does not model actual hardware timing, pipelines, or concurrent race conditions.
- **Reliability**: Minimal bounds checking and limited error recovery mechanisms compared to production kernels.

# 6 Conclusion

This Memory Management Simulator provides a robust environment for studying the relationship between high-level memory allocation and low-level hardware caching. Through the use of hybrid data structures and a hierarchical access model, the project successfully demonstrates how modern operating systems and hardware interact to manage resources efficiently while minimizing performance bottlenecks.