Memory Management Simulator

Overview

The Memory Management Simulator is a modular system designed to model the life cycle of memory in a modern operating system. It simulates physical memory allocation, multi-level cache hierarchies, and virtual memory management using paging and translation buffers aside.

Memory Layout and Assumptions

• **Physical Address Space**: 1024 bytes, organized into 16 frames.

• **Virtual Address Space**: 4096 bytes, organized into 64 pages.

• **Page/Frame Size**: Each page and frame is 64 bytes in size.

• **Alignment:** 8-byte boundary

- Enforced in the linear allocator to simulate realistic hardware constraints
- Prevents unaligned memory access penalties

• **Block Representation:**

- Memory is managed as a list of blocks
- Each block contains a header with:
  - Unique identifier (ID)
  - Allocated size
  - Requested size
  - Allocation status (free/used)

• **TLB:** 16-entry, 4-way set-associative (4 sets) caching VPN→PFN translations using LRU

• **Page Replacement:** LRU algorithm (configurable)

• **Cache Hierarchy:** L1 (64Bytes), L2 (256Bytes), L3 (512Bytes) with LRU replacement (configurable)

Allocation Strategy Implementations

Linear Allocator (Memory Allocator)

The **Linear Allocator** manages memory using a **doubly linked list of blocks**, where each block represents a contiguous memory region that is either **allocated** or **free**.

It supports multiple allocation strategies:

- **First-Fit**: Allocates the first free block large enough for the request.
- **Best-Fit**: Allocates the smallest suitable free block to minimize internal fragmentation.
- **Worst-Fit**: Allocates the largest free block to avoid creating very small fragments.

Upon deallocation, the allocator performs **coalescing** by merging the freed block with adjacent free blocks in the list. This reduces **external fragmentation** and improves allocation efficiency.

Buddy System Design

The **Buddy System** allocator improves allocation efficiency by organizing memory into blocks whose sizes are always **powers of two** (2^k).

Power-of-Two Alignment: *All memory requests are* ***rounded up to the next power of two****. This ensures that every allocation fits exactly into a valid buddy block size, simplifying block management and merging.*

Splitting Algorithm:

If a request is made for a block of size 2^k and only a larger block is available, the allocator repeatedly **splits the larger block into two equal-sized “buddy” blocks**. This splitting continues recursively until the smallest possible power-of-two block that can satisfy the request is obtained.

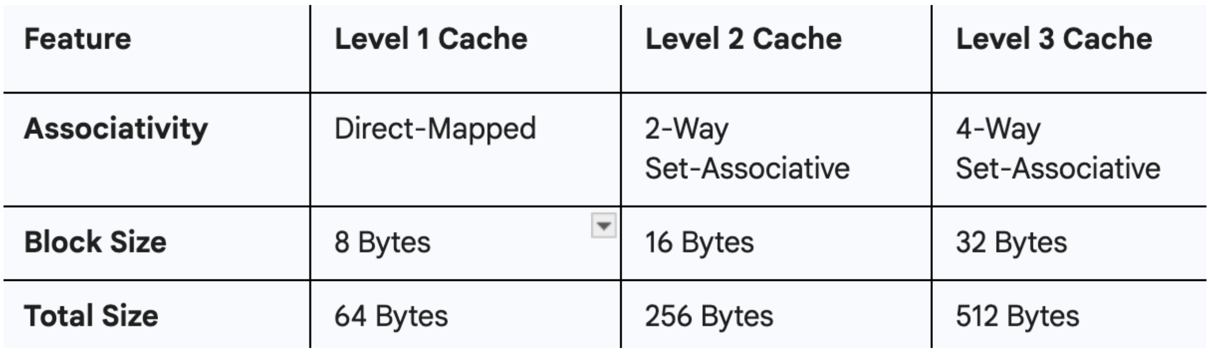
Exclusive-OR (XOR) Merge Logic:

When a block is freed, the allocator computes the address of its corresponding **buddy block** using a **bitwise Exclusive-OR (XOR)** operation between the block’s starting address and its size.
If the buddy block is also free, both blocks are **merged back into a single larger**

block. This merging process may continue recursively, allowing memory to be recombined into larger blocks when possible.

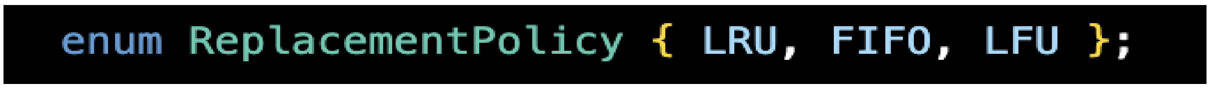
Cache hierarchy and replacement policy

The simulator models a **configurable three-level cache hierarchy** that closely reflects modern CPU cache design.
Each cache level varies in size, block granularity, and associativity, while the **replacement policy is selectable at runtime**, allowing flexible experimentation with different eviction strategies.



Feature	Level 1 Cache	Level 2 Cache	Level 3 Cache
Associativity	Direct-Mapped	2-Way Set-Associative	4-Way Set-Associative
Block Size	8 Bytes	16 Bytes	32 Bytes
Total Size	64 Bytes	256 Bytes	512 Bytes

Replacement policies:



```
enum ReplacementPolicy { LRU, FIFO, LFU };
```

Replacement Policy Implementations

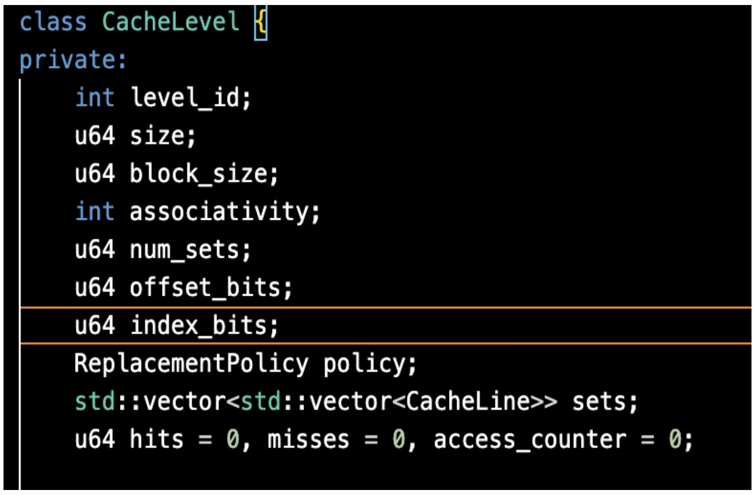
Least Recently Used (LRU)
Evicts the cache line that has not been accessed for the longest duration.
Each cache line maintains a last_access_time, updated on every hit or insertion.
Eviction selects the line with the **oldest timestamp** within the set.

First-In First-Out (FIFO)
Evicts the cache line that was inserted earliest into the cache.
Each cache line records a loaded_time at insertion, which remains unchanged on hits.
The line with the **earliest insertion time** is selected for eviction.

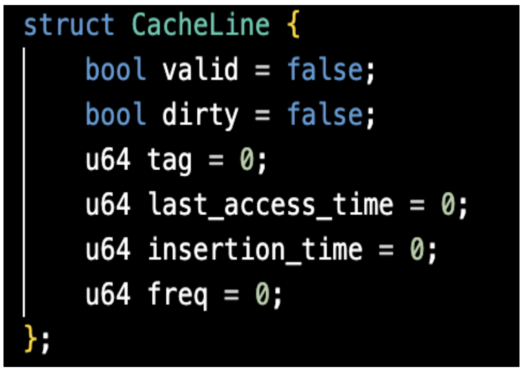
Least Frequently Used (LFU):

The Least Frequently Used (LFU) replacement policy evicts the cache line that has been accessed the fewest number of times.

Each cache line maintains an access_count, which is incremented on every cache hit. When a replacement is required, the cache line with the lowest access_count is selected for eviction. If multiple cache lines have the same access frequency, a tiebreaker is applied. In such cases, the least recently used cache line—determined using its last access time—is evicted.



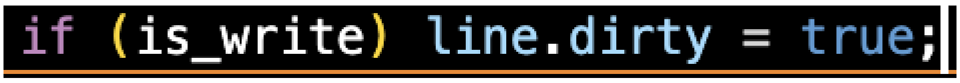
```
class CacheLevel {
private:
    int level_id;
    u64 size;
    u64 block_size;
    int associativity;
    u64 num_sets;
    u64 offset_bits;
    u64 index_bits;
    ReplacementPolicy policy;
    std::vector<std::vector<CacheLine>> sets;
    u64 hits = 0, misses = 0, access_counter = 0;
```



```
struct CacheLine {
    bool valid = false;
    bool dirty = false;
    u64 tag = 0;
    u64 last_access_time = 0;
    u64 insertion_time = 0;
    u64 freq = 0;
};
```

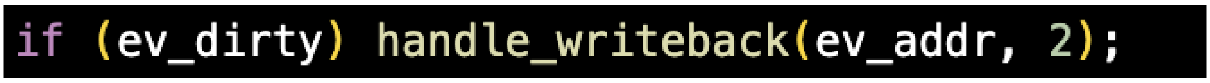
Write-Back Policy (with Write-Back Flow)

On a write operation, the cache updates only the current cache line and marks it as dirty:



```
if (is_write) line.dirty = true;
```

No immediate write is performed to the next cache level or main memory.
The modified data is written back **only when a dirty line is evicted**:

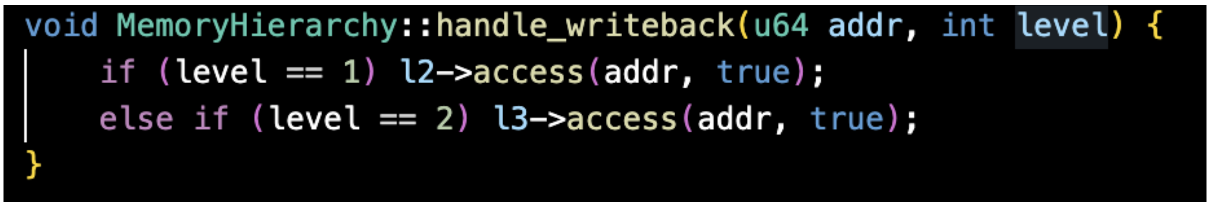


```
if (ev_dirty) handle_writeback(ev_addr, 2);
```

Write-Back Flow:

- L1 eviction → write back to L2
- L2 eviction → write back to L3
- L3 eviction → write back to main memory (simulated)

This delayed write mechanism minimizes memory traffic and improves overall cache performance.



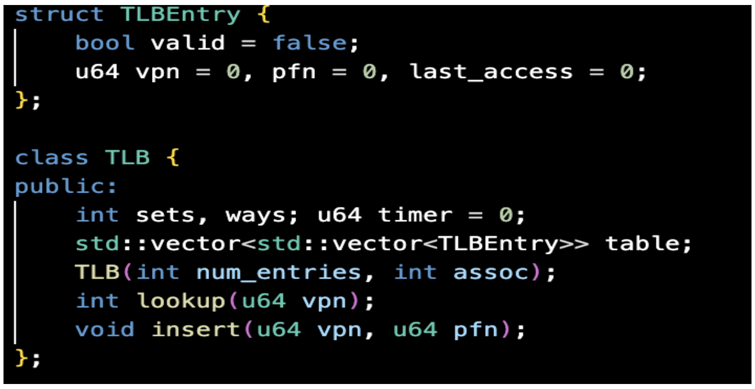
```
void MemoryHierarchy::handle_writeback(u64 addr, int level) {
    if (level == 1) l2->access(addr, true);
    else if (level == 2) l3->access(addr, true);
}
```

Virtual memory model

The Virtual Memory subsystem translates **64-bit virtual addresses** into **physical memory addresses**. It uses a Translation Lookaside Buffer (TLB), a page table, and a page replacement mechanism to manage memory efficiently and reduce disk accesses.

Translation Lookaside Buffer (TLB):

The TLB is a **4-way set-associative cache** that stores recent virtual-to-physical page translations, reducing the overhead of page table lookups. It uses a **set associative Least Recently Used (LRU)** policy to evict the least recently accessed entry within a set.

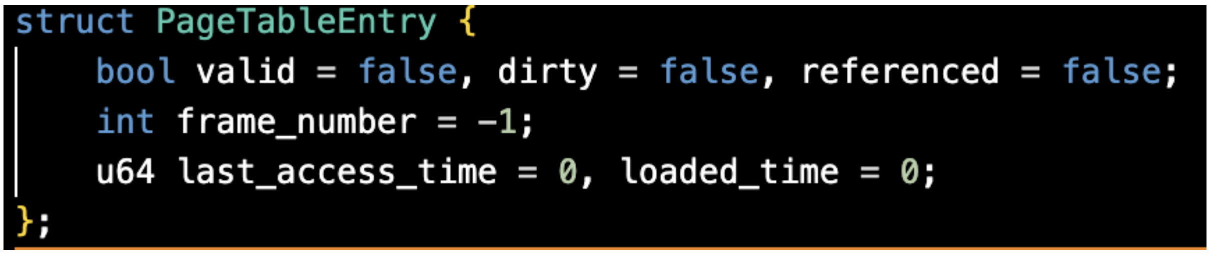


```
struct TLBEntry {
    bool valid = false;
    u64 vpn = 0, pfn = 0, last_access = 0;
};

class TLB {
public:
    int sets, ways; u64 timer = 0;
    std::vector<std::vector<TLBEntry>> table;
    TLB(int num_entries, int assoc);
    int lookup(u64 vpn);
    void insert(u64 vpn, u64 pfn);
};
```

Page Table:

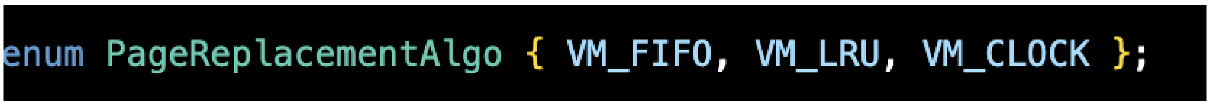
The page table is implemented as a **flat vector indexed by the Virtual Page Number (VPN)**. Each entry maintains a **Valid bit** to indicate residency in physical memory and a **Dirty bit** to track modifications. Additional metadata supports page replacement decisions.



```
struct PageTableEntry {
    bool valid = false, dirty = false, referenced = false;
    int frame_number = -1;
    u64 last_access_time = 0, loaded_time = 0;
};
```

Page Replacement Algorithms

The Virtual Memory subsystem supports multiple page replacement algorithms, selectable using the following enumeration:



```
enum PageReplacementAlgo { VM_FIFO, VM_LRU, VM_CLOCK };
```

When no page replacement algorithm is set at runtime, LRU is selected by default

- **VM_FIFO (First-In First-Out):**
  Evicts the page that has been resident in physical memory for the longest time, based on its load order.
- **VM_LRU (Least Recently Used):**
  Evicts the page with the oldest access timestamp, representing the page least recently accessed.
- **VM_CLOCK:**
  Implements the Clock (Second Chance) algorithm, using a circular pointer and reference bits to approximate LRU with lower overhead.

Address Translation Flow

1. **Virtual Page Number and Offset**
   The virtual address is divided into a Virtual Page Number (VPN) and an Offset using bitwise shifts and masks.

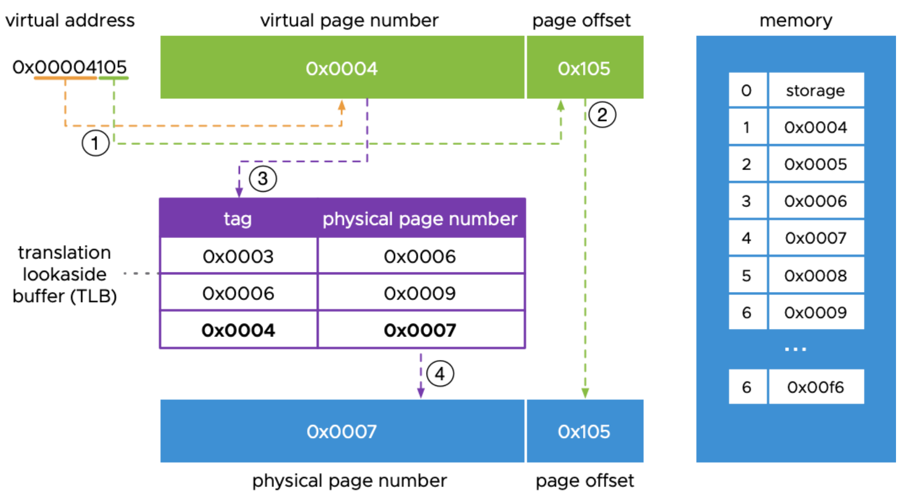
$$\text{Offset} = VA \ \& \ (2^k - 1)$$

$$\text{VPN} = VA \gg k$$

2. **Translation Buffer Lookup**
   The system first searches the Translation Lookaside Buffer (TLB) for a cached VPN-to-Physical Frame Number (PFN) mapping.
3. **Page Table Walk**
   If the TLB lookup misses, the system consults the Page Table to locate the corresponding physical frame.
4. **Physical Address Calculation**
   The Physical Address is computed by multiplying the Physical Frame Number by the page size and adding the offset.

$$PA = (\text{PFN} \times \text{Page Size}) + \text{Offset}$$

5. **Cache System Integration**
   The resulting physical address is sent to the Level 1 (L1) cache to begin data access.



Limitations and simplifications

- **Fixed Cache and Virtual Memory Configuration**
  Cache levels (L1, L2, L3), cache sizes, replacement policies, and virtual/physical memory sizes are fixed and small compared to real systems.
- **Simplified Memory Management Algorithms**
  Allocation and page replacement use basic algorithms (Buddy, FIFO, LRU, CLOCK) without advanced techniques such as compaction, adaptive replacement, or access permissions.
- **Memory latency is ignored:** The simulator does not model cycle-accurate timing for cache or memory accesses, whereas real CPUs have precise and varying latencies for cache hits and misses across the memory hierarchy.
- **Paging Structure Limitation**
  The system uses **only basic single-level (normal) paging** for virtual memory management. It does not implement more advanced paging schemes such as **hierarchical (multi-level) page tables**, **inverted page tables**, or **hashed paging**.

- **Internal Fragmentation Not Handled**

  The system does not explicitly handle internal fragmentation. Memory is allocated in fixed-size pages and cache blocks, and any unused space within an allocated page or block is wasted. The simulator does not track or optimize this unused space.