# Design and Implementation of a Memory Management Simulator


# – Suvrat Mehta (23116097)


## 1. Memory Layout and Assumptions

The simulator models physical memory not as a raw byte array, but as a logical collection of memory blocks managed via a **Doubly Linked List** data structure.

### Memory Representation

- **Logical Units:** Memory is simulated as a contiguous block of integers ranging from `0` to `Memory Size - 1`. It does not store actual user data payloads, but rather tracks the *occupancy status* of address ranges.
- **Block Structure:** The memory is divided into nodes (`Block`), where each node represents a contiguous chunk of memory. The `Block` structure defined in `memory.hpp` contains:
    - `StartAddress`: The integer offset where the block begins.
    - `Size`: Length of the block.
    - `Id`: Unique identifier for the allocation (0 if free).
    - `Hole`: Boolean flag (`true` = Free, `false` = Allocated).
    - `NextBlock` / `PrevBlock`: Pointers to adjacent memory chunks.

### Assumptions

- **Contiguity:** Physical memory is treated as a single linear address space initialized as one large "Hole".
- **Addressing:** Addresses are absolute integers starting at 0.
- **Overhead:** The simulation assumes "perfect" metadata storage; the `Block` headers themselves do not consume simulated memory space (internal fragmentation is strictly defined by the difference between requested and allocated size, though currently in my project allocations are exact matches to requests).

## 2. Allocation Strategy Implementations

The simulator implements three dynamic memory allocation strategies. All strategies utilize the linked list to traverse memory segments.

### First Fit

- **Algorithm:** The allocator traverses the linked list from the `Head`. It selects the **first** hole found that is equal to or larger than the requested `size`.

- **Implementation:** * If the found hole is larger than the request, it is split: the current node is resized to the requested size, and a new "Hole" node is created for the remaining space.
    - If no suitable block is found, the allocation fails.

### Best Fit

- **Algorithm:** The allocator traverses the **entire** list to identify the hole that is smallest among those large enough to satisfy the request.
- **Implementation:**
    - Minimizes wasted space by picking the "tightest" fit.
    - Like First Fit, it splits the chosen block if there is excess space.

### Worst Fit

- **Algorithm:** The allocator traverses the **entire** list to identify the **largest** available hole.
- **Implementation:**
    - Intended to leave large enough gaps for future allocations after splitting.
    - Splits the largest block found.

### 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 simulates a configurable multilevel cache hierarchy (Level 1 and Level 2) integrated with the memory allocator.

### Cache Structure

- **Organization:** The cache is **Set-Associative**.
    - Calculated as `Num_Sets = Size / (Associativity * Block_Size)`.
- **Addressing:** Memory addresses are decoded using bitwise operations:
    - **Offset:** `address & ((1 << offset_bits) - 1)`
    - **Index:** `(address >> offset_bits) & ((1 << index_bits) - 1)`
    - **Tag:** `address >> (offset_bits + index_bits)`
- **Storage:** Implemented as a `vector` of `deque<CacheLine>`, where each deque represents a Set.

### Replacement Policy: FIFO

- **Policy: First-In, First-Out (FIFO)**.

- **Implementation:**
  - New blocks are inserted at the **front** of the deque (`push_front`).
  - When a set reaches capacity (`size == Associativity`), the block at the **back** of the deque is removed (`pop_back`).

### Hierarchy Flow

1. **Read L1:** Check L1 Cache. If Hit, return.
2. **Read L2:** If L1 Miss, check L2.
   - If L2 Hit: Load block into L1.
1. **Memory Access:** If L2 Miss, fetch from Main Memory.
   - **Write-Allocate:** The block is loaded into **both** L2 and L1 caches.

# 4. Limitations and Simplifications

The following trade-offs were made to balance complexity with the project's educational goals:

1. **Data Abstraction:** The simulator does not store actual application data. A `malloc(100)` reserves 100 units of space but does not provide a buffer for writing real bytes.
2. **Integer Addressing:** Addresses are simple integers rather than hexadecimal pointers, simplifying arithmetic and debugging.
3. **Cache Coherence:** The simulation assumes a single-core environment; therefore, cache coherence protocols (like MESI) are not implemented.
4. **Replacement Policies:** The cache implements FIFO.
5. **Memory Alignment:** Memory is assumed to be perfectly aligned; the simulator does not enforce word-boundary alignment constraints typically found in hardware.