# Memory Management — Design & Code References

## 1. Memory Layout and Assumptions

The system simulates a contiguous main memory space represented as a linear address range starting from address `0` up to `totalMemory - 1`.

### Assumptions

- Memory is byte-addressable
- Allocation sizes are given in bytes
- No paging or disk I/O is simulated
- Memory requests are sequential and single-threaded
- Cache addresses are logical addresses mapped directly to main memory

For non-buddy allocation strategies, memory is modeled as a linked list of blocks, where each block contains:

- Start address
- Size
- Allocation ID
- Free/used status

### Code Reference

```
struct Memory::Block{
    int start, size, id;
    bool free;
    Block* next;
};
```

Memory is initialized as a single free block:

```
head = new Block(0, size, -1, true, nullptr);
```

---

## 2. Allocation Strategy Implementations

The simulator supports four allocation strategies, selectable at runtime.

### Runtime Selection

```
enum class AllocatorType{
    FIRST_FIT, BEST_FIT, WORST_FIT, BUDDY
};
```

```
bool Memory::setAllocator(std::string& type){
    if (type == "first_fit") allocator = AllocatorType::FIRST_FIT;
    else if (type == "best_fit") allocator = AllocatorType::BEST_FIT;
    else if (type == "worst_fit") allocator = AllocatorType::WORST_FIT;
    else if (type == "buddy") allocator = AllocatorType::BUDDY;
}
```

---

### 2.1 First Fit

- Traverses the block list from the beginning
- Allocates the first free block that fits

```
int Memory::mallocFF(int need){
    Block* cur = head;
    while (cur){
        if (cur->free && cur->size >= need)
            return allocate(cur, need);
        cur = cur->next;
    }
    return -1;
}
```

---

### 2.2 Best Fit

- Searches all free blocks
- Chooses the smallest sufficient block

```
int Memory::mallocBF(int need){
    Block* cur = head, *best = nullptr;
    while (cur){
        if (cur->free && cur->size >= need)
            if (!best || best->size > cur->size)
                best = cur;
        cur = cur->next;
    }
    return best ? allocate(best, need) : -1;
}
```

---

### 2.3 Worst Fit

- Chooses the largest available free block

```
```

```
int Memory::mallocWF(int need){
    Block* cur = head, *worst = nullptr;
    while (cur){
        if (cur->free && cur->size >= need)
            if (!worst || worst->size < cur->size)
                worst = cur;
        cur = cur->next;
    }
    return worst ? allocate(worst, need) : -1;
}
```

---

### Common Behavior

- Blocks are split if larger than required
- Adjacent free blocks are merged on deallocation
- Allocation IDs uniquely identify allocations

```
void Block::splitAndAllocate(int need, int id){
    Block* split = new Block(start + need, size - need, -1, true, next);
    size = need;
    free = false;
    this->id = id;
    next = split;
}

void Block::mergeNext(){
    Block* tmp = next;
    size += tmp->size;
    next = tmp->next;
    delete tmp;
}
```

---

## 3. Buddy System Design

The buddy allocator manages memory in power-of-two sized blocks.

### Design

- Total memory divided into blocks of size `2^k`
- Each block's buddy is computed using XOR
- Free blocks tracked using free lists per order

### Data Structures

```
```

```
std::vector<std::vector<int>> freeLists;
std::unordered_map<int,std::pair<int,int>> buddyAllocated;
```

---

## Allocation

```
int Memory::buddyMalloc(int size){
    int need = 1;
    while (need < size) need <<= 1;
}
```

### Block Splitting

```
while (cur > order){
    cur--;
    int buddy = id + (1 << cur);
    freeLists[cur].push_back(buddy);
}
```

---

## Deallocation

### Buddy Calculation

```
int buddy = id ^ (1 << order);
```

### Recursive Merge

```
while (order < maxOrder){
    auto it = std::find(list.begin(), list.end(), buddy);
    if (it == list.end()) break;
    list.erase(it);
    id = std::min(id, buddy);
    order++;
}
```

---

## Fragmentation Tracking

```
internalFrag += allocSize - size;
```

## 4. Cache Hierarchy and Replacement Policy

The cache subsystem supports multi-level caches via chaining.

```
Cache* next;    // Next cache level
Memory* memory; // Backing memory
```

### Cache Structure

- Set-associative cache
- Configurable size, block size, associativity

```
numBlocks = cacheSize / blockSize;
numSets   = numBlocks / associativity;
```

### Address Format

```
[ Tag | Index | Offset ]
```

```
int blockNumber = address / blockSize;
int index = blockNumber % numSets;
int tag   = blockNumber / numSets;
```

### Replacement Policies

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

Policy-specific metadata:

```
int lastUsed;
int frequency;
int insertedAt;
```

Victim selection example (LRU):

```
if (line.lastUsed < victim->lastUsed)
    victim = &line;
```

---

## 4.1 Cache Invalidation on Memory Deallocation

To maintain correctness between the cache hierarchy and main memory, the system implements **cache line invalidation** when a memory region is freed.

### Rationale

When a memory block is deallocated, any cache lines corresponding to that memory region may contain **stale data**. To prevent invalid accesses, all cache lines overlapping the freed address range are invalidated.

This simplified model assumes:

- No dirty bits
- No write-back or write-through policies
- Invalidation is sufficient for correctness

---

### Invalidation Strategy

- The address range `[start, start + size)` is computed
- Each cache set and way is scanned
- Cache lines whose memory range overlaps the freed region are invalidated
- Invalidation is propagated recursively to lower cache levels

---

### Address Overlap Check

For each valid cache line:

- The starting address of the cached block is reconstructed using its tag and index
- If the cache block overlaps the freed memory range, the line is invalidated

---

### Cache Invalidation Code

```
void Cache::invalidateRange(int start, int size){
    int end = start + size;
    for (int i = 0; i < numSets; i++){
        for (int j = 0; j < associativity; j++){
            if (!sets[i][j].valid) continue;

            int blockStart = (sets[i][j].tag * numSets + i) * blockSize;
            int blockEnd = blockStart + blockSize;
```

```
            if (blockStart < end && blockEnd > start)
                sets[i][j].valid = false;
        }
    }

    if (next) next->invalidateRange(start, size);
}
```

---

### Design Notes

- Invalidation is **range-based**, not ID-based
- Works uniformly for all cache levels
- Avoids the complexity of coherence protocols
- Suitable for a single-threaded simulation model

---

## 5. Address Translation Flow

```
CPU Address
   ↓
Cache L1
   ↓
Cache L2
   ↓
Main Memory
```

### Code Flow

```
if (next) next->access(address);
else if (memory) memory->access(address);
```

---

### Address Interpretation Across Cache Levels

Each cache level independently interprets the same byte address using its own block size and indexing scheme. The simulator forwards the original byte address between cache levels rather than derived block identifiers.

This design models cache *tag lookup behavior* instead of cache line fill granularity. As a result, different cache levels may use different block sizes without affecting correctness, since each cache only determines whether it contains data covering the requested address.

---

## 6. Statistics and Reporting

### Memory Statistics

```
std::cout << "Used memory : " << usedMemory << '\n';
std::cout << "Internal fragmentation : " << internalFrag << '\n';
```

### Cache Statistics

```
std::cout << "Hits : " << hits << '\n';
std::cout << "Misses : " << misses << '\n';
```

---

## 7. Limitations and Simplifications

- No multithreading
- No paging or swapping
- No dirty bits or write policies
- No cache coherence
- No timing simulation

These choices keep the focus on core memory management and cache behavior.

---

## 8. Summary

This system provides:

- Dynamic memory allocation (FF, BF, WF, Buddy)
- Buddy allocator with fragmentation tracking
- Multi-level cache simulation
- Configurable replacement policies
- Detailed statistics and reporting

---

## 9. System Initialization and Configuration

The simulator initializes the entire memory and cache hierarchy at startup based on user-provided parameters, with safe defaults applied when inputs are omitted or invalid.

### Initialization Responsibilities

- Configure main memory size and allocation strategy
- Enforce allocator-specific constraints (e.g., buddy allocator requires power-of-two memory size)
- Configure the cache hierarchy (L1 and L2 caches)
- Validate cache parameters to preserve correct address decoding
- Apply default configurations on invalid input to avoid undefined states

### Design Rationale

- Memory and cache sizes are fixed at initialization to preserve address consistency
- Cache parameters are restricted to powers of two to allow clean tag–index–offset decomposition
- Partial reconfiguration is intentionally disallowed to maintain memory–cache invariants
- Invalid user inputs fall back to defaults to keep the simulator robust and reproducible

---

### 9.1 Cache Configuration Constraints

Cache parameters must satisfy the following constraints:

- Cache size, block size, and associativity must be strictly positive
- Cache size must be divisible by block size
- The number of blocks must be divisible by associativity
- All cache parameters must be powers of two

These constraints reflect real hardware requirements and ensure correct address decoding.

---

### Code Reference: Cache Configuration Validation

```
bool validCacheConfig(int cacheSize, int blockSize, int associativity) {
    if (cacheSize <= 0 || blockSize <= 0 || associativity <= 0)
        return false;

    if (cacheSize % blockSize != 0)
        return false;

    int numBlocks = cacheSize / blockSize;
    if (numBlocks % associativity != 0)
        return false;

    if (!isPowerOfTwo(cacheSize) ||
        !isPowerOfTwo(blockSize) ||
        !isPowerOfTwo(associativity))
        return false;

    return true;
}
```

---

### 9.2 Buddy Allocator Constraint

When the buddy allocation strategy is selected, the total memory size must be a power of two. This constraint is necessary to preserve correct buddy alignment and enable recursive splitting and merging using XOR-based buddy computation.

If a non-power-of-two memory size is provided while selecting the buddy allocator, the system automatically falls back to a non-buddy allocation strategy to avoid invalid memory states.

```
if (allocType == "buddy" && !isPowerOfTwo(memSize)) {
    allocType = "first_fit";
}
```

---

## 9.3 Default System Configuration

| Component | Default |
|---|---|
| Main memory size | 1024 bytes |
| Allocation strategy | first_fit |
| L1 cache | 64 B, block size 16 B, 2-way |
| L2 cache | 256 B, block size 16 B, 4-way |
| Cache replacement policy | FIFO |

---

### 9.4 Cache Hierarchy Ordering Constraint

The simulator enforces a strict cache hierarchy:

- L1 cache size < L2 cache size < main memory size

This constraint preserves the semantic meaning of a cache hierarchy. If violated during initialization, the system reinitializes using default parameters instead of attempting partial correction, ensuring all cache invariants remain valid.

---

## 10. System Reinitialization

The simulator supports full system reinitialization through a dedicated `reinit` command.

### Reinitialization Semantics

- All memory and cache structures are destroyed
- Cache contents and statistics are cleared
- The system is reconfigured from scratch using the same initialization flow
- Partial reinitialization (memory-only or cache-only) is intentionally not supported

This behavior is semantically equivalent to restarting the simulator and preserves correctness across memory and cache subsystems.

---

## 11. Testing and Validation

The simulator was validated using a comprehensive interactive test workload designed to exercise all major components of the system under realistic usage scenarios.

### Test Methodology

Validation was performed via an interactive command-line session that simulates:

- Memory allocation and deallocation patterns
- Cache access behavior across multiple cache levels
- Policy changes during execution
- Fragmentation behavior for different allocators
- Full system reinitialization

The workload intentionally includes both valid and invalid configurations to verify error handling and fallback behavior.

### Test Artifacts

A complete input–output transcript of the test workload is provided separately:
`tests/sample_input_workload_with_expected_output.txt`

This file serves as:

- The sample input workload
- Cache and virtual address access log
- Expected output and correctness reference

Each command in the workload is followed by the observed simulator output, which is used to verify correctness.

### Correctness Criteria

The simulator is considered correct if:

- Allocation success or failure matches allocator constraints
- Cache hit/miss behavior aligns with the active replacement policy
- Cache lines overlapping freed memory regions are invalidated
- Fragmentation statistics reflect allocator-specific behavior
- Reinitialization restores the system to a clean state

### Notes on Address Semantics

Cache accesses operate on virtual memory addresses within allocated regions.
Allocation identifiers are used solely as handles for deallocation and are not treated as memory addresses, except in the case of the buddy allocator where the returned identifier corresponds to the block's start address.