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);
```

---

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