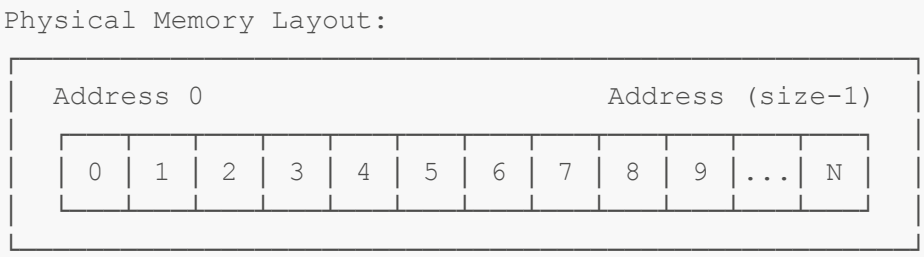
Memory Management Simulator - Design Document

Table of Contents

1. Memory Layout and Assumptions

1.1 Physical Memory Model

Structure: Contiguous byte array of configurable size

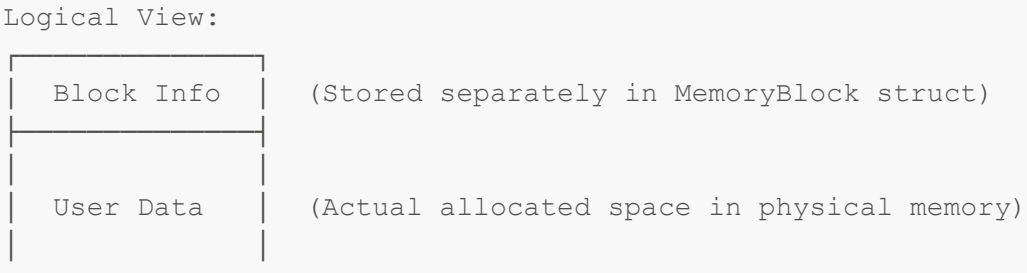


Key Properties:

- Byte-addressable (1-byte granularity)
- Bounds-checked access
- No alignment requirements enforced
- No protection mechanisms

1.2 Block Metadata Storage

Block metadata is stored separately from allocated memory.



```
  └─────────────────┘

Actual Implementation:
                     MemoryBlock struct          Physical Memory
                     (not in physical mem)       (user data only)
                     ┌──────────────────┐        ┌──────────────────┐
                     │ start_address    │──────->│                  │
                     │ size             │        │  User Data       │
                     │ is_free          │        │                  │
                     │ id               │        └──────────────────┘
                     │ next             │
                     │ prev             │
                     └──────────────────┘
```

Rationale: Simplifies simulation and makes reported sizes exact.

1.3 Key Assumptions

1. Single-threaded execution
2. Instantaneous operations (no timing simulation)
3. Perfect hardware (no bit errors)
4. Write-through cache policy
5. Zero-time page faults (page faults still counted, but no latency penalty)
6. No memory protection or segmentation
7. No TLB (Translation Lookaside Buffer)
8. Cache misses count as events (for statistics), but have no timing penalty

---

2. Allocation Strategy Implementations

2.1 Data Structure

Doubly-linked list of memory blocks

```
Memory Block List:

HEAD
  |
  v
┌─────────────────┐     ┌─────────────────┐     ┌─────────────────┐
│  Free Block     │<--->│ Allocated (1)   │<--->│  Free Block     │
│  addr: 0        │     │  addr: 1024     │     │  addr: 2048     │
│  size: 1024     │     │  size: 1024     │     │  size: 14336    │
│  is_free: T     │     │  is_free: F     │     │  is_free: T     │
└─────────────────┘     └─────────────────┘     └─────────────────┘
```

2.2 First Fit

Algorithm:

```
function allocate(size):
    for each block in free_list:
        if block.size >= size:
            split_block_if_larger(block, size)
            return block.start_address
    return ERROR
```

Characteristics:

- Fast allocation
- Tends to fragment beginning of memory
- Time Complexity: O(n)

2.3 Best Fit

Algorithm:

```
function allocate(size):
    best_block = null
    min_waste = infinity

    for each block in free_list:
        if block.size >= size:
            waste = block.size - size
            if waste < min_waste:
                best_block = block
                min_waste = waste

    if best_block:
        split_block_if_larger(best_block, size)
        return best_block.start_address
    return ERROR
```

Characteristics:

- Minimizes wasted space per allocation
- Can create many small unusable fragments
- Time Complexity: O(n)

2.4 Worst Fit

Algorithm:

```
function allocate(size):
    worst_block = null
    max_size = 0

    for each block in free_list:
```

```
        if block.size >= size AND block.size > max_size:
            worst_block = block
            max_size = block.size

    if worst_block:
        split_block_if_larger(worst_block, size)
        return worst_block.start_address
    return ERROR
```

Characteristics:

- Leaves larger leftover fragments
- Can delay fragmentation issues
- Time Complexity: O(n)

2.5 Block Splitting

```
Before Split:
┌──────────────────────────────────┐
│      Free Block (4096)           │
└──────────────────────────────────┘

After allocate(1024):
┌────────────────┐┌──────────────────┐
│ Allocated      ││   Free           │
│   (1024)       ││   (3072)         │
└────────────────┘└──────────────────┘
```

2.6 Block Coalescing

Adjacent free blocks merge on deallocation:

```
Before Deallocation:
┌────────────┐┌──────────────┐┌────────────┐
│   Free     ││ Allocated    ││   Free     │
│  (1024)    ││   (1024)     ││  (2048)    │
└────────────┘└──────────────┘└────────────┘

After deallocate(middle_block):
┌──────────────────────────────────────────┐
│           Free (4096)                    │
└──────────────────────────────────────────┘
```

---

3. Buddy System Design

3.1 Block Structure

Power-of-2 sized blocks with recursive splitting:

```
Buddy Block Memory Layout (1024 bytes):

Level 0: ┌─────────────────────────────────────────┐
         │          1024 bytes                     │
         └─────────────────────────────────────────┘

Level 1: ┌──────────────────────┐┌──────────────────────┐
         │    512 bytes         ||    512 bytes       │
         └──────────────────────┘└──────────────────────┘

Level 2: ┌──────────┐┌──────────┐┌──────────┐┌──────────┐
         │  256     ||  256     ||  256     ||  256     │
         └──────────┘└──────────┘└──────────┘└──────────┘

... down to minimum block size (32 bytes)
```

3.2 Free Lists Organization

Data Structure: Map of size to list of free blocks

```
free_lists_:

  Size 1024: [addr: 0] -> NULL

  Size 512:  NULL

  Size 256:  NULL

  Size 128:  NULL

  Size 64:   [addr: 128] -> [addr: 256] -> NULL

  Size 32:   [addr: 512] -> NULL
```

3.3 Buddy Address Calculation

Formula: `buddy_address = address XOR size`

Example:

```
Block at address 0x0000, size 128:
Buddy address = 0x0000 XOR 128 = 0x0080

Block at address 0x0080, size 128:
Buddy address = 0x0080 XOR 128 = 0x0000
```

```
Binary:
  0x0000 = 0000 0000 0000 0000
  0x0080 = 0000 0000 1000 0000
  XOR    = 0000 0000 1000 0000 = 0x0080
```

This proves the XOR property: each block's buddy is computed using XOR.

3.4 Allocation Algorithm

```
function allocate(requested_size):
    # Check bounds
    if requested_size > max_block_size:
        return ERROR

    size = max(round_up_to_power_of_2(requested_size), min_block_size)

    # Try exact size
    if free_lists_[size] not empty:
        return free_lists_[size].pop_front()

    # Find larger block and split
    larger_size = size * 2
    while larger_size <= max_size:
        if free_lists_[larger_size] not empty:
            block_addr = free_lists_[larger_size].pop_front()
            # Split repeatedly until we get desired size
            # One half becomes allocated, other half goes to free list
            split_recursively(block_addr, larger_size, size)
            return block_addr
        larger_size *= 2

    return ERROR
```

3.5 Splitting Process

```
Split 256-byte block into two 128-byte buddies:

Before:
┌──────────────────────────────────┐
│     Block at 0x0000 (256)        │
└──────────────────────────────────┘

After:
┌──────────────────┐ ┌──────────────────┐
│ 0x0000 (128)     ‖ 0x0080 (128)     │
│   (returned)     ‖   (to free list)│
└──────────────────┘ └──────────────────┘
                     ^
                     Buddy address = 0x0000 XOR 0x0080
```

3.6 Coalescing Process

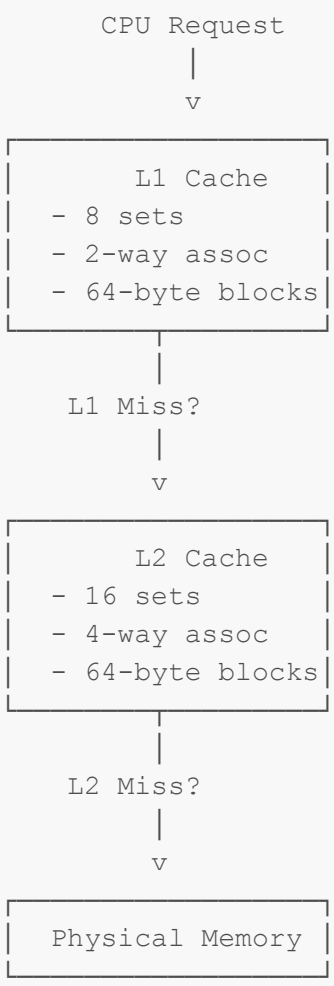
```
function deallocate(block):
    buddy_addr = block.address XOR block.size
    buddy = find_buddy_in_free_list(buddy_addr, block.size)

    if buddy exists:
        remove buddy from free_list
        merged_addr = min(block.address, buddy_addr)
        merged_size = block.size * 2
        deallocate(merged_block)  # Recursive merge
    else:
        add block to free_list[block.size]
```

---

4. Cache Hierarchy and Replacement Policy

4.1 Architecture

Two-level cache hierarchy:



4.2 Set-Associative Organization

Example: 4-way set-associative cache with 8 sets

```
         Set 0    Set 1    Set 2    ...    Set 7
        ┌──────┐ ┌──────┐ ┌──────┐       ┌──────┐
Way 0   │Line  │ │Line  │ │Line  │       │Line  │
        ├──────┤ ├──────┤ ├──────┤       ├──────┤
Way 1   │Line  │ │Line  │ │Line  │       │Line  │
        ├──────┤ ├──────┤ ├──────┤       ├──────┤
Way 2   │Line  │ │Line  │ │Line  │       │Line  │
        ├──────┤ ├──────┤ ├──────┤       ├──────┤
Way 3   │Line  │ │Line  │ │Line  │       │Line  │
        └──────┘ └──────┘ └──────┘       └──────┘
```

4.3 Address Decomposition

Formula: `set = (address / block_size) % num_sets`

For 64-byte blocks, 8 sets:

```
Physical Address Breakdown:

 31                                    6  5    3  2        0
┌──────────────────────────────────────┬──────┬──────────────┐
│              Tag                     │  Set │   Offset     │
│          (23 bits)                   │(3 bits)│ (6 bits)   │
└──────────────────────────────────────┴──────┴──────────────┘
                                                  │
                                                  └─> Byte within block
                                           │
                                           └─> Which set to check
                  │
                  └─> Stored in cache line for comparison
```

4.4 Cache Access Flow

```
function cache_read(address):
    [tag, set_index, offset] = parse_address(address)

    # Check all ways in the set
    for way in sets_[set_index]:
        if way.valid AND way.tag == tag:
            way.recordAccess(global_time)
            return way.data[offset]  # HIT

    # MISS: Load from lower level
    stats.misses++
```

```
    victim_way = selectVictim(set_index)
    loadBlock(address, tag, set_index, victim_way)
    return sets_[set_index][victim_way].data[offset]

Cache Policy:
- Read-allocate: Read misses load blocks into cache
- Write-allocate: Write misses also load blocks into cache
- Write-through: All writes go directly to memory
```

4.5 Replacement Policy (LRU)

Least Recently Used:

```
function selectVictim(set_index):
    # First, check for invalid (empty) lines
    for way in sets_[set_index]:
        if NOT way.valid:
            return way  # Always use empty line first

    # All lines valid, use LRU
    oldest_time = infinity
    victim = 0
    for way in sets_[set_index]:
        if way.last_access < oldest_time:
            oldest_time = way.last_access
            victim = way

    return victim

Example:
Time 0:  Access A -> last_access = 0
Time 5:  Access B -> last_access = 5
Time 10: Access A -> last_access = 10 (updated)
Time 15: Need eviction -> Evict B (smallest time = 5)
```

4.6 Write-Through Policy

All writes go to memory immediately:

```
function cache_write(address, data):
    # Write to memory first
    memory.write(address, data)

    # Check if block is in cache
    line = findLine(set_index, tag)
    if line exists:
        # Cache hit - update cache line
        line.data[offset] = data
        line.recordAccess(global_time)
```

```
    else:
        # Cache miss - load block and update (write-allocate)
        victim_way = selectVictim(set_index)
        loadBlock(address, tag, set_index, victim_way)
        sets_[set_index][victim_way].data[offset] = data
```

Key property: Write-through with write-allocate. Both read and write misses load blocks into cache.

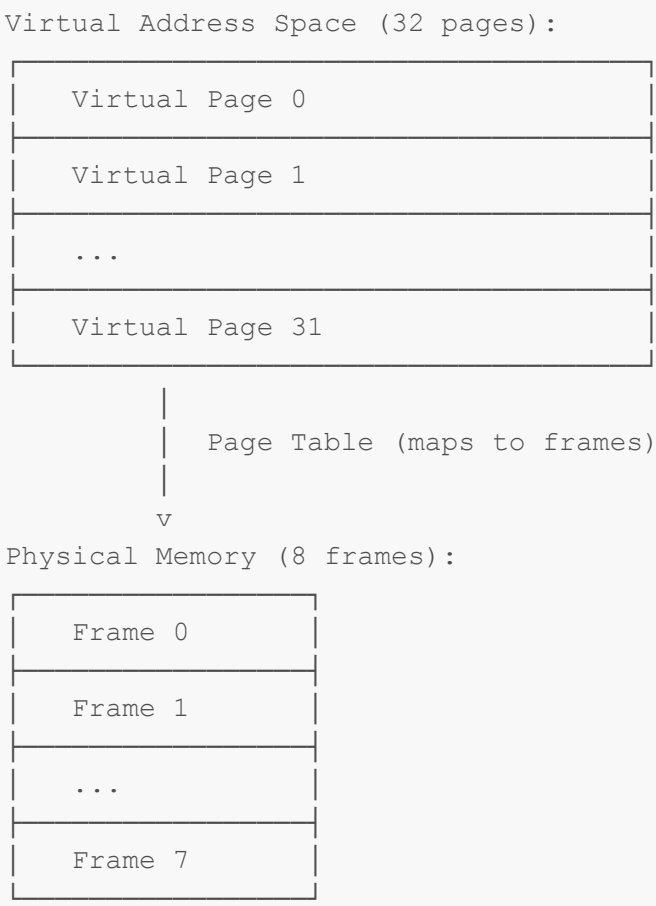
---

5. Virtual Memory Model

5.1 Address Space Layout

Single-level page table mapping virtual pages to physical frames.

Page Table: Statically allocated with 32 entries, initially all marked invalid.



5.2 Virtual Address Format

Example: 512-byte pages, 32 virtual pages

```
┌──────────────────┬──────────────────┐
│   (5 bits)       │   (9 bits)       │
└──────────────────┴──────────────────┘
      0-31            0-511

Extraction:
  offset = address & 0x1FF           # Lower 9 bits
  page_number = address >> 9          # Upper 5 bits
```

5.3 Physical Address Construction

```
After translation:
┌──────────────────┬──────────────────┐
│  Frame Number    │  Page Offset     │
│  (from PTE)      │  (from VA)       │
└──────────────────┴──────────────────┘

Construction:
  physical_address = (frame_number << offset_bits) | offset
```

5.4 Page Fault Handling

```
function translate(virtual_address):
    [page_number, offset] = parseAddress(virtual_address)
    pte = page_table_[page_number]

    if pte.valid:
        # Page is in memory
        stats.page_hits++
        pte.recordAccess(global_time)
        return constructPhysicalAddress(pte.frame_number, offset)
    else:
        # PAGE FAULT
        stats.page_faults++

        # Find free frame or select victim
        frame = findFreeFrame()
        if frame == NOT_FOUND:
            frame = selectVictimPage()
            evictPage(frame)

        # Load page into frame
        loadPage(page_number, frame)

        # Update page table
        pte.valid = true
        pte.frame_number = frame
        pte.load_time = global_time
        pte.last_access = global_time
```

```
        return constructPhysicalAddress(frame, offset)
```

5.5 Page Replacement Policy (LRU)

Least Recently Used:

```
function selectVictimFrame():
    oldest_access = infinity
    victim_frame = 0

    # Inspect each physical frame to find which page it contains
    for each frame in physical_memory:
        page = find_page_mapped_to_frame(frame)
        if page_table_[page].last_access < oldest_access:
            oldest_access = page_table_[page].last_access
            victim_frame = frame

    return victim_frame
```

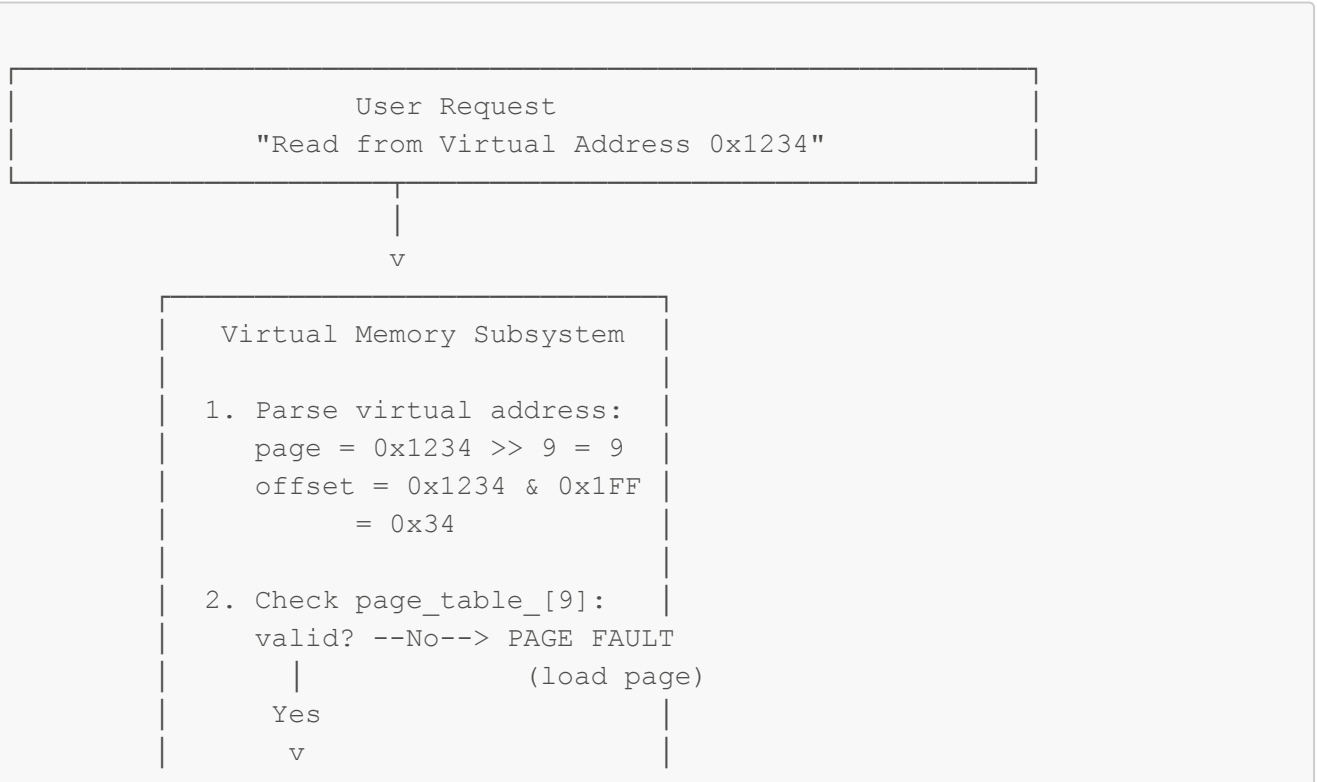
Critical property: LRU evicts the page with smallest last_access time, NOT FIFO (which would evict by load_time).

The victim frame is selected by inspecting which page is mapped to each frame via the page table.

---

6. Address Translation Flow

6.1 Complete Translation Path



```
          |  3. Get frame = 3           |
          |  4. Construct:              |
          |     phys_addr = (3 << 9) | 0x34
          |               = 0x634       |
          └──────────────┬──────────────┘
                         |
                         | Physical Address: 0x634
                         v
          ┌──────────────────────────────┐
          |   Cache Hierarchy            |
          |                              |
          |  1. Parse address:           |
          |     tag = 0x634 >> 6         |
          |     set = (0x634 / 64) % 8   |
          |     offset = 0x634 & 0x3F    |
          |                              |
          |  2. Check L1 cache[set]:     |
          |     Hit? --Yes--> Return data
          |       |                      |
          |      No                      |
          |       v                      |
          |  3. Check L2 cache:          |
          |     Hit? --Yes--> Return data
          |       |                      |
          |      No                      |
          └──────┬┴──────────────────────┘
                 |
                 v
          ┌──────────────────────────────┐
          |   Physical Memory            |
          |                              |
          |  1. Read from address 0x634  |
          |  2. Return byte value        |
          |  3. Load into caches         |
          └──────────────────────────────┘
```

6.2 Independence of Subsystems

Each subsystem operates independently:

```
Virtual Memory:
  - Knows about: Virtual addresses, page tables, physical memory
  - Does NOT know about: Cache hierarchy

Cache Hierarchy:
  - Knows about: Physical addresses, cache lines, physical memory
  - Does NOT know about: Virtual memory or page tables

Physical Memory:
  - Knows about: Byte array and addresses
  - Does NOT know about: Virtual memory or cache
```

7. Limitations and Simplifications

7.1 What This Simulator Does NOT Model

Real Performance:

- No actual timing (hits/misses counted, but no latency)
- No instruction-level parallelism
- No out-of-order execution

Concurrency:

- Single-threaded only
- No locks or race conditions
- No multi-core coherence protocols

Memory Protection:

- No segmentation or permissions
- No MMU protection checks
- Any address can access any memory

Hardware Realism:

- No prefetching
- No speculative execution
- No TLB (Translation Lookaside Buffer)
- No bit errors or ECC

Advanced Features:

- No huge pages
- No multi-level page tables
- No copy-on-write

7.2 Valid Use Cases

Educational Purposes:

- Understanding allocation algorithms
- Learning cache replacement policies
- Studying page replacement behavior
- Exploring locality effects
- Comparing strategy trade-offs

Algorithm Comparison:

- Which fit strategy has less fragmentation?
- Which cache policy has better hit ratio?
- How does working set size affect page faults?

7.3 Invalid Use Cases

Performance Prediction:

- Cannot predict real-world performance
- Cannot estimate actual latency
- Cannot benchmark production systems

System Sizing:

- Cannot determine optimal cache size for workload
- Cannot predict actual memory requirements
- Results do not translate to real hardware

7.4 Design Trade-offs

Choice	Benefit	Cost
External metadata	Exact user sizes, simple code	Not realistic
Write-through + write-allocate	Memory always consistent, simpler implementation	Cache pollution from write-only data
Single-level page table	Simple O(1) translation	High memory overhead
No TLB	Easier to understand	Unrealistic hit rates
Instant operations	Simple simulation	No performance model

7.5 Physical Memory Ownership

Important: The allocator and virtual memory subsystems operate in **separate modes**:

Mode 1: Allocator Mode

- Allocator manages physical memory directly
- Virtual memory is disabled
- Physical addresses used directly

Mode 2: Virtual Memory Mode

- Virtual memory system owns physical frames
- Allocator is disabled
- All addresses are virtual and translated to physical frames

These modes are mutually exclusive - they cannot be used simultaneously in the same simulation. The simulator allows either allocator-based memory management OR virtual memory with paging, but not both at once.

7.6 Implementation Notes

Allocator Switching Behavior:

- Switching between allocation strategies (e.g., First Fit → Buddy) creates a new allocator instance
- All previous allocations are invalidated when switching allocators
- Block IDs restart from 1 with each allocator change
- The simulator warns users when switching: "Warning: Switching allocator. All previous allocations invalidated."

Fragmentation Metrics:

- **Standard Allocators** (First/Best/Worst Fit):
  - Internal fragmentation: Wasted space within allocated blocks
  - External fragmentation: Free memory that cannot satisfy allocation requests
- **Buddy Allocator**:
  - Internal fragmentation: Overhead from rounding to power-of-2 sizes
  - "Buddy fragmentation (unusable free)": Percentage of free memory that is not in the largest free block
  - Note: This metric differs from classic external fragmentation as buddy systems have minimal external fragmentation by design

Error Handling:

- Invalid block IDs return: "Block ID not found (allocator may have been reset or invalid ID)"
- This helps distinguish between truly invalid IDs and IDs from previous allocator instances

---