# Memory Simulator

Design Document

# Contents

# 1 Overview

This Memory Management Simulator is an educational tool that demonstrates core operating system concepts related to memory management. It provides hands-on simulation of:

- **Dynamic Memory Allocation**: First Fit, Best Fit, and Worst Fit strategies
- **Memory Fragmentation**: External fragmentation tracking and visualization
- **Cache Simulation**: Multi-level (L1/L2) cache with configurable parameters
- **Cache Replacement Policies**: FIFO and LRU implementations
- **Write Policies**: Write-back with write-allocate, dirty bit tracking

The simulator is implemented in C++17 and provides an interactive command-line interface for experimenting with different configurations.

# 2 System Architecture

## 2.1 Module Structure

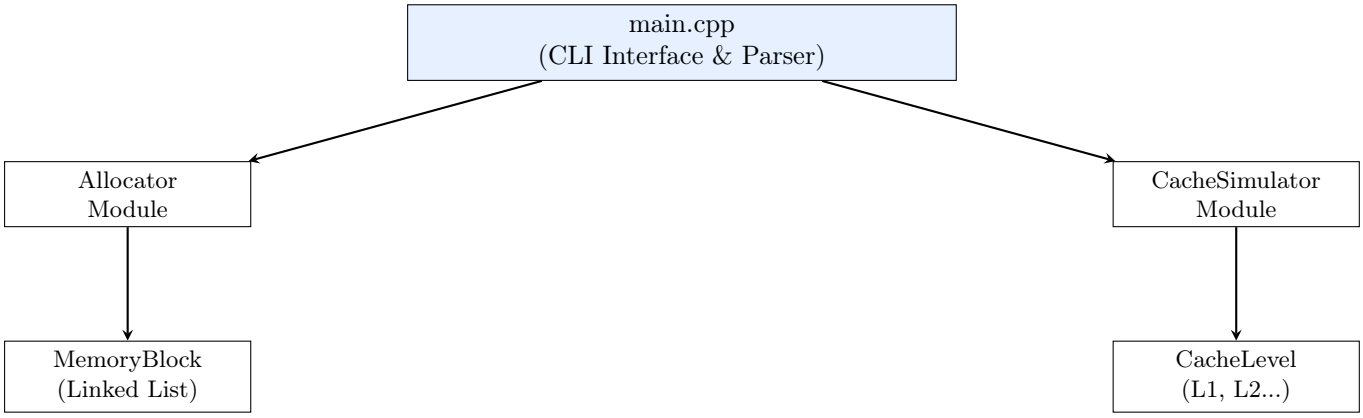


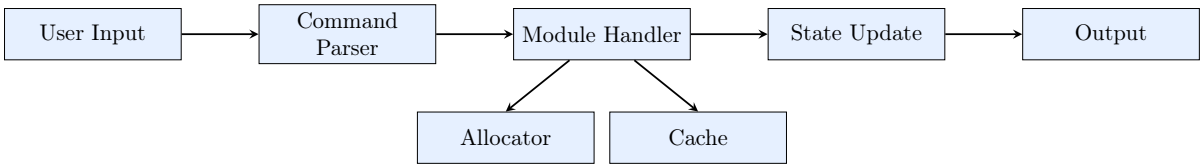

Figure 1: System Architecture Module Structure

## 2.2 File Organization

Project Directory Structure

```
memory-simulator/
├── include/
│   ├── allocator.h ..... Allocator class declaration
│   ├── cache.h ..... Cache simulator declarations
│   └── memory_block.h ..... MemoryBlock struct
├── src/
│   ├── main.cpp ..... CLI interface
│   ├── allocator/
│   │   └── allocator.cpp ..... Allocation strategy implementations
│   └── cache/
│       └── cache.cpp ..... Cache simulation logic
├── tests/
│   └── workload*.txt ..... Test workload files
└── docs/
    └── design doc.pdf ..... This document
```

## 2.3 Data Flow




Workload files can be piped: `./build/memsim < workload.txt`

Figure 2: Data Flow Diagram

# 3 Memory Layout and Assumptions

## 3.1 Memory Model

The simulator models physical memory as a contiguous block managed through a **doubly-linked list** of memory blocks:

```
struct MemoryBlock {
    size_t address;      // Starting byte address
    size_t size;         // Block size in bytes
    bool is_free;        // Allocation status
    int block_id;        // Unique ID (-1 if free)
    MemoryBlock* next;   // Next block pointer
    MemoryBlock* prev;   // Previous block pointer
};
```

Listing 1: MemoryBlock Structure

### 3.2 Memory Layout Visualization

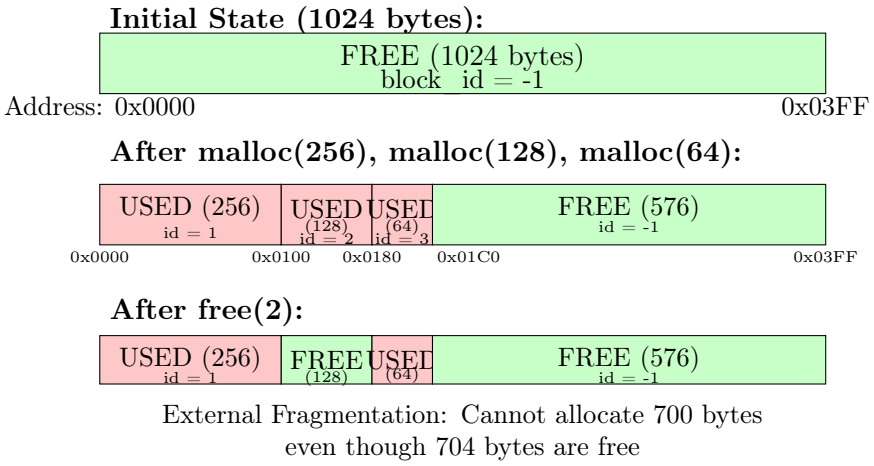



Figure 3: Memory Layout Evolution

### 3.3 Design Assumptions

| Assumption | Rationale |
|---|---|
| Byte-addressable memory | Simplifies allocation math; real systems use word alignment |
| No alignment requirements | Educational simplicity; real allocators align to 4/8/16 bytes |
| Instant allocation | No simulation of allocation time; focus on strategy behavior |
| Unique block IDs | Monotonically increasing; simplifies tracking |
| No memory protection | No simulation of access rights or segmentation |

Table 1: Memory Model Design Assumptions

### 3.4 Why Doubly-Linked List?

**Advantages**

- ✓ O(1) block splitting: Just adjust pointers
- ✓ O(1) coalescing: Can access both neighbors
- ✓ Dynamic sizing: No pre-defined block count
- ✓ Memory efficient: Only metadata overhead per block

**Trade-offs**

- × O(n) search for allocation (traversal required)
- × Pointer overhead per block (16 bytes on 64-bit)

## 4 Allocation Strategy Implementations

## 4.1 First Fit

**Algorithm:**

```
findBlock(size):
    for each block in list (from head):
        if block.is_free AND block.size >= size:
            return block
    return NULL
```

**Implementation:**

```
MemoryBlock* Allocator::firstFit(size_t size) {
    MemoryBlock* current = head;
    while (current != nullptr) {
        if (current->is_free && current->size >= size) {
            return current;
        }
        current = current->next;
    }
    return nullptr;
}
```

Listing 2: First Fit Implementation

**Characteristics:**

| Metric | Value |
|---|---|
| Time Complexity | O(n) worst, O(1) best |
| Fragmentation | Tends to fragment start of memory |
| Best For | General purpose, fast allocation |

Table 2: First Fit Characteristics

## 4.2 Best Fit

**Algorithm:**

```
findBlock(size):
    best = NULL
    for each block in list:
        if block.is_free AND block.size >= size:
            if best == NULL OR block.size < best.size:
                best = block
    return best
```

**Implementation:**

```
MemoryBlock* Allocator::bestFit(size_t size) {
    MemoryBlock* best = nullptr;
    MemoryBlock* current = head;
    while (current != nullptr) {
        if (current->is_free && current->size >= size) {
            if (best == nullptr || current->size < best->size) {
                best = current;
            }
```

```
        }
        current = current->next;
    }
    return best;
}
```

Listing 3: Best Fit Implementation

**Characteristics:**

| Metric | Value |
|---|---|
| Time Complexity | O(n) always (must scan all) |
| Fragmentation | Creates many small unusable fragments |
| Best For | Memory-constrained systems |

Table 3: Best Fit Characteristics

## 4.3 Worst Fit

**Algorithm:**

```
findBlock(size):
    worst = NULL
    for each block in list:
        if block.is_free AND block.size >= size:
            if worst == NULL OR block.size > worst.size:
                worst = block
    return worst
```

**Implementation:**

```
MemoryBlock* Allocator::worstFit(size_t size) {
    MemoryBlock* worst = nullptr;
    MemoryBlock* current = head;
    while (current != nullptr) {
        if (current->is_free && current->size >= size) {
            if (worst == nullptr || current->size > worst->size) {
                worst = current;
            }
        }
        current = current->next;
    }
    return worst;
}
```

Listing 4: Worst Fit Implementation

**Characteristics:**

| Metric | Value |
|---|---|
| Time Complexity | O(n) always |
| Fragmentation | Leaves larger remaining fragments |
| Best For | Workloads with similar-sized allocations |

Table 4: Worst Fit Characteristics

## 4.4 Block Splitting

When a block is larger than the requested size, it is split:

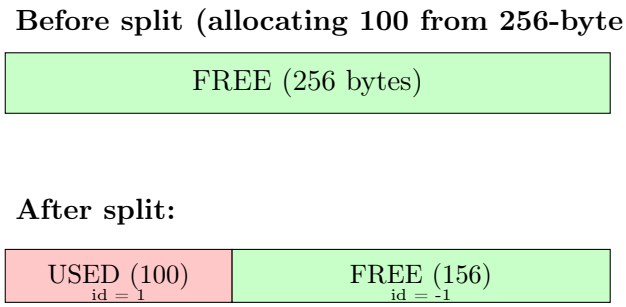



Figure 4: Block Splitting Example

**Implementation:**

```
if (block->size > size) {
    MemoryBlock* new_free = new MemoryBlock(
        block->address + size,     // New address
        block->size - size,        // Remaining size
        true, -1                   // Free, no ID
    );
    // Update linked list pointers...
    block->size = size;
}
```

Listing 5: Block Splitting

## 4.5 Memory Coalescing

When a block is freed, adjacent free blocks are merged to reduce fragmentation:

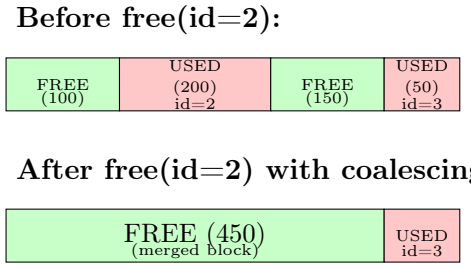



Figure 5: Memory Coalescing Example

**Implementation:**

```
void Allocator::coalesce(MemoryBlock* block) {
    // Merge with next blocks
    while (block->next && block->next->is_free) {
        MemoryBlock* next = block->next;
        block->size += next->size;
        block->next = next->next;
        if (next->next) next->next->prev = block;
        delete next;
    }
    // Merge with previous blocks
```

```
    while (block->prev && block->prev->is_free) {
        MemoryBlock* prev = block->prev;
        prev->size += block->size;
        prev->next = block->next;
        if (block->next) block->next->prev = prev;
        delete block;
        block = prev;
    }
}
```

Listing 6: Memory Coalescing

### 4.6 Fragmentation Metrics

**External Fragmentation Formula:**

$$\text{External Fragmentation} = \left(1 - \frac{\text{Largest Free Block}}{\text{Total Free Memory}}\right) \times 100\% \tag{1}$$

**Example:**

**Memory State:**

[USED 100][FREE 50][USED 200][FREE 150][USED 100][FREE 100]

Total Free = 50 + 150 + 100 = 300 bytes
Largest Free Block = 150 bytes

External Fragmentation = $(1 - 150/300) \times 100\% = 50\%$

**Interpretation:** We have 300 bytes free, but cannot allocate anything larger than 150 bytes.

## 5 Cache Hierarchy and Replacement Policy

### 5.1 Cache Architecture

The simulator implements a **multi-level set-associative cache** with configurable parameters:

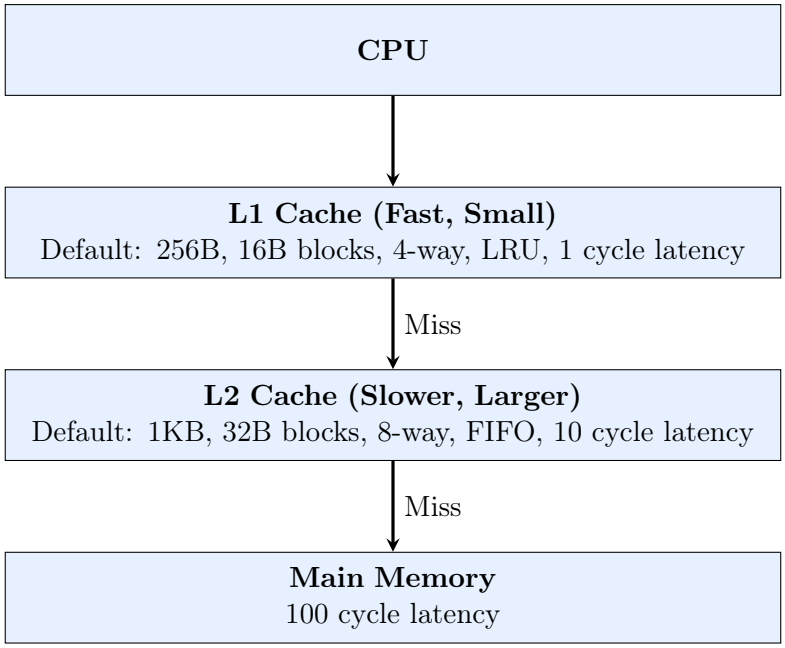



Figure 6: Cache Hierarchy

## 5.2 Cache Line Structure

```
struct CacheLine {
    bool valid;           // Contains valid data?
    bool dirty;           // Modified since loaded? (for write-back)
    size_t tag;           // Address tag for matching
    size_t last_access;   // Access counter for LRU
};
```

Listing 7: Cache Line Structure

**Valid Bit:** Indicates if the cache line contains meaningful data.

- `valid = false`: Line is empty or invalidated
- `valid = true`: Line contains cached data

**Dirty Bit:** Tracks if data has been written (modified).

- `dirty = false`: Data matches main memory (clean)
- `dirty = true`: Data modified, memory is stale

**Tag:** Upper address bits used to identify which memory block is cached.

## 5.3 Address Decomposition

For a cache with block size B and S sets:

**Address Bits:**


Figure 7: Address Bit Decomposition

**Example:** 32-bit address, 16B blocks, 4 sets

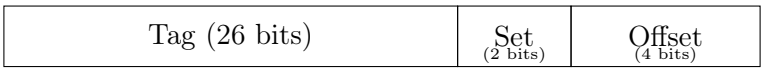



Figure 8: 32-bit Address Example

**Implementation:**

```
size_t CacheLevel::getSetIndex(size_t address) const {
    size_t block_bits = (size_t)std::log2(block_size);  // Offset bits
    return (address >> block_bits) % num_sets;
}

size_t CacheLevel::getTag(size_t address) const {
    size_t block_bits = (size_t)std::log2(block_size);
    size_t set_bits = (size_t)std::log2(num_sets);
    return address >> (block_bits + set_bits);
}
```

Listing 8: Address Decomposition Functions

## 5.4 Set-Associative Organization

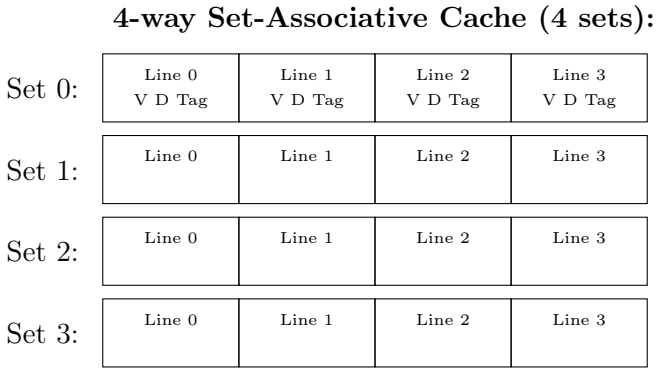



Figure 9: 4-way Set-Associative Cache Organization

**Cache Mappings:**

- **Direct-Mapped:** associativity = 1 (1 line per set)
- **Fully Associative:** num_sets = 1 (all lines in one set)
- **N-way Set-Assoc:** N lines per set

## 5.5 Replacement Policies

### 5.5.1 FIFO (First-In-First-Out)

Evicts the oldest entry in the set, regardless of access pattern.

Implementation: Queue per set tracking insertion order

```
Insert A → Insert B → Insert C → Insert D (cache full)
Queue: [A, B, C, D]

Insert E (miss, need to evict):
  - Evict A (front of queue)
  - Queue: [B, C, D, E]
```

**Data Structure:**

```
std::vector<std::list<size_t>> fifo_queues;  // One queue per set

// On insertion:
fifo_queues[set_index].push_back(line_index);

// On eviction:
size_t victim = fifo_queues[set_index].front();
fifo_queues[set_index].pop_front();
```

Listing 9: FIFO Implementation

### 5.5.2 LRU (Least Recently Used)

Evicts the line that hasn't been accessed for the longest time.

Implementation: Counter-based tracking

```
Access A (counter=1) → Access B (2) → Access C (3) → Access A (4)
Line A: last_access = 4
Line B: last_access = 2  ← LRU victim (smallest counter)
Line C: last_access = 3
```

**Implementation:**

```
// On every access:
set[line].last_access = ++access_counter;

// Finding victim:
size_t victim = 0;
size_t min_access = set[0].last_access;
for (size_t i = 1; i < associativity; i++) {
    if (set[i].last_access < min_access) {
        min_access = set[i].last_access;
        victim = i;
    }
}
```

Listing 10: LRU Implementation

## 5.6 Write Policy Implementation

The simulator implements **Write-Back with Write-Allocate**:

| Policy | Behavior |
|---|---|
| **Write-Back** | Writes go only to cache; memory updated on eviction |
| **Write-Allocate** | On write miss, load block into cache then write |

Table 5: Write Policy Summary

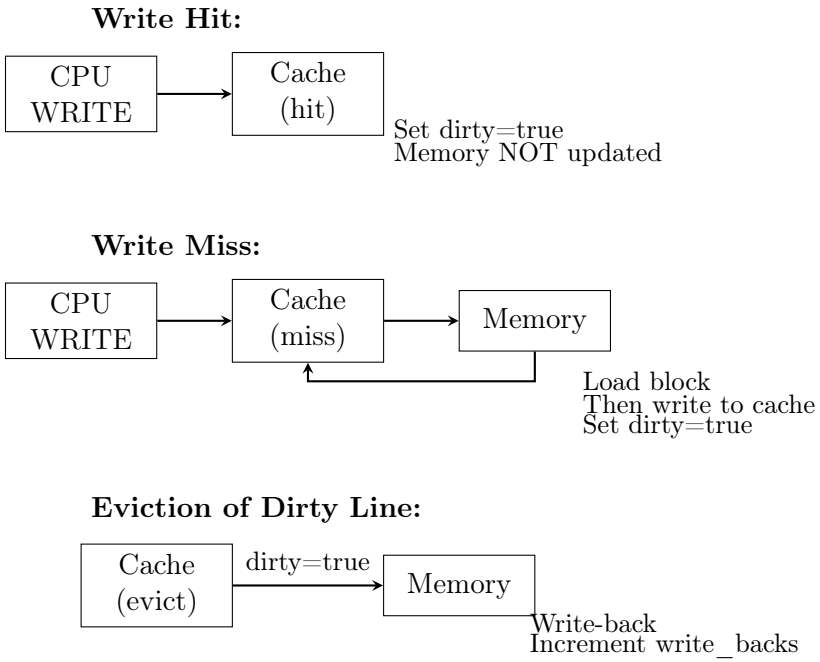



Figure 10: Write Policy Behavior

**Implementation:**

```
bool CacheLevel::access(size_t address, bool isWrite) {
    // ... hit detection ...

    if (hit) {
        if (isWrite) {
            set[i].dirty = true;  // Mark as modified
        }
        return true;
    }

    // Miss - find victim
    size_t victim = findVictim(set_index);

    // Write-back if dirty
    if (set[victim].valid && set[victim].dirty) {
        stats.write_backs++;  // Would write to memory
    }

    // Load new line
    set[victim].dirty = isWrite;  // Dirty if this is a write
    // ...
}
```

Listing 11: Write Policy Implementation

### 5.7 Cache Statistics

The simulator tracks comprehensive statistics:

```
struct CacheStats {
    size_t hits;              // Cache hits
    size_t misses;            // Cache misses
    size_t accesses;          // Total accesses
    size_t write_backs;       // Dirty evictions
    size_t total_access_time; // Cumulative latency

    double hitRatio() const {
        return accesses > 0 ? (double)hits / accesses * 100.0 : 0.0;
    }
};
```

Listing 12: Cache Statistics Structure

**Sample Output:**

```
=== Cache Statistics ===
L1:
  Accesses:    100
  Hits:        75
  Misses:      25
  Write-backs: 8
  Hit Rate:    75.00%
  Access Time: 100 cycles
L2:
  Accesses:    25
  Hits:        15
  Misses:      10
  Write-backs: 3
  Hit Rate:    60.00%
  Access Time: 250 cycles
------------------------
Total Access Time: 1350 cycles
Memory Latency:    100 cycles
========================
```

## 6 Address Translation Flow

## 6.1 Cache Access Flow

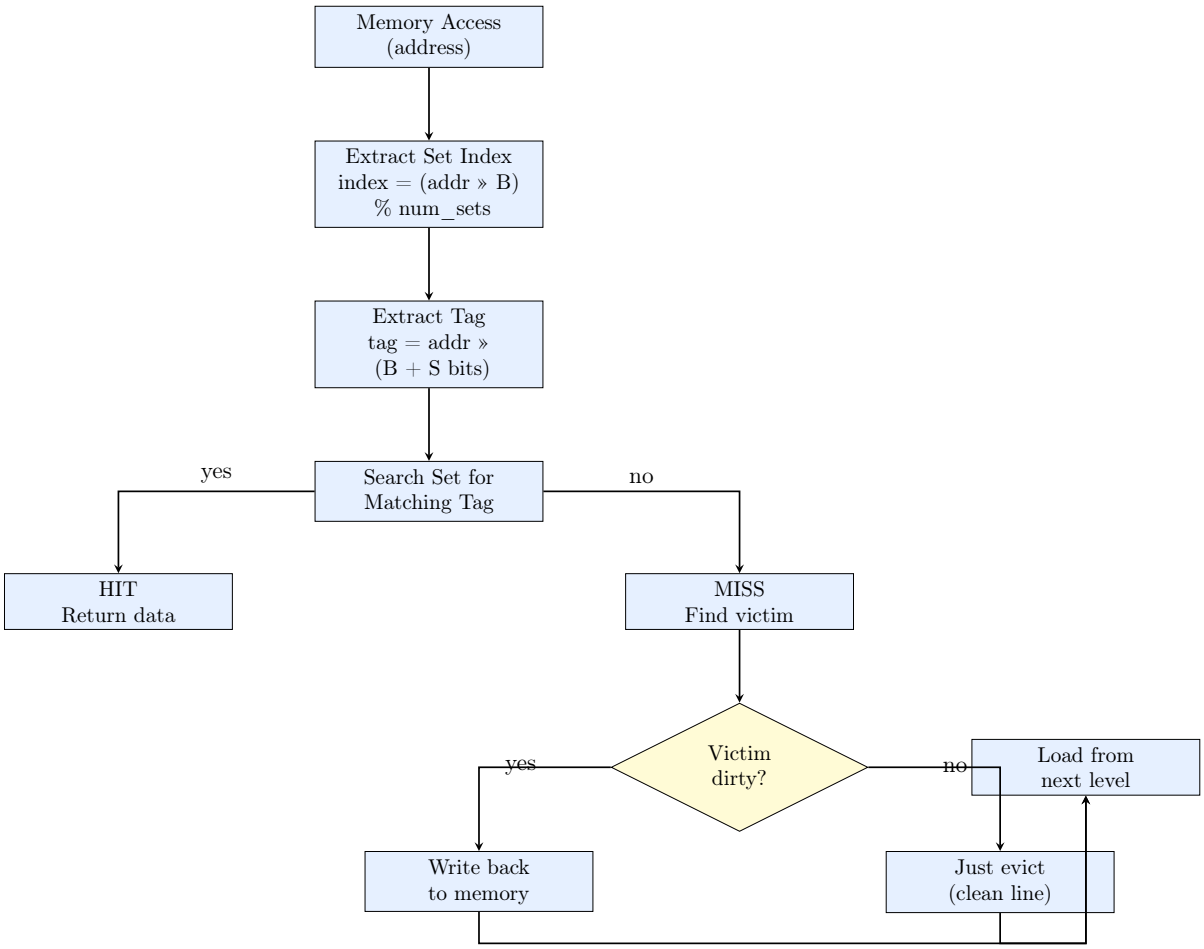



Figure 11: Cache Access Flow Diagram

### 6.2 Multi-Level Access Example

Access address 0x1234:

**Step 1: Check L1**

```
Set Index = (0x1234 >> 4) % 4 = 2
Tag = 0x1234 >> 6 = 0x48
→ Search L1 Set 2... MISS
```

**Step 2: Check L2**

```
Set Index = (0x1234 >> 5) % 8 = 1
Tag = 0x1234 >> 8 = 0x12
→ Search L2 Set 1... MISS
```

**Step 3: Access Main Memory**

```
→ Fetch block containing 0x1234
```

**Step 4: Install in caches**

```
→ Load into L2, then L1
```

**Total Latency:** L1(1) + L2(10) + Memory(100) = **111 cycles**

## 7 Limitations and Simplifications

### 7.1 Memory Allocator Limitations

| Limitation | Real-World Behavior |
|---|---|
| No alignment | Real allocators align to word/cache-line boundaries |
| No metadata overhead | Real malloc stores size, flags in block headers |
| Instant operations | Real systems have timing for memory operations |
| No thread safety | Real allocators handle concurrent access |
| No memory protection | Real systems have page-level access control |

Table 6: Memory Allocator Limitations

### 7.2 Cache Simulator Limitations

| Limitation | Real-World Behavior |
|---|---|
| Exclusive hierarchy | Most caches are inclusive (L1 $\subseteq$ L2) |
| No actual data | Only addresses tracked, no data storage |
| Single CPU | No cache coherence protocols (MESI, etc.) |
| No prefetching | Real caches speculatively load adjacent blocks |
| Simplified timing | Real timing includes bus contention, queue delays |
| Power-of-2 only | Block sizes and set counts must be powers of 2 |

Table 7: Cache Simulator Limitations

### 7.3 What We Simulate vs. Don't Simulate

| ✓ Simulated | × Not Simulated |
|---|---|
| Allocation strategies | Virtual memory/paging |
| Fragmentation tracking | TLB (Translation Lookaside Buffer) |
| Cache hit/miss behavior | Cache coherence |
| Replacement policies | Prefetching |
| Write-back policy | Bus arbitration |
| Latency accumulation | Actual data storage |
| Dirty bit tracking | Memory protection |

Table 8: Simulation Coverage

## 8 Future Extensions

### 8.1 Buddy System Allocator

Power-of-2 block sizes with efficient splitting/coalescing:

Example: 1024 bytes total

**Request 100 bytes:**

```
1024 → split → 512 + 512
512  → split → 256 + 256
256  → split → 128 + 128
Allocate 128-byte block (smallest power of 2 >= 100)
```

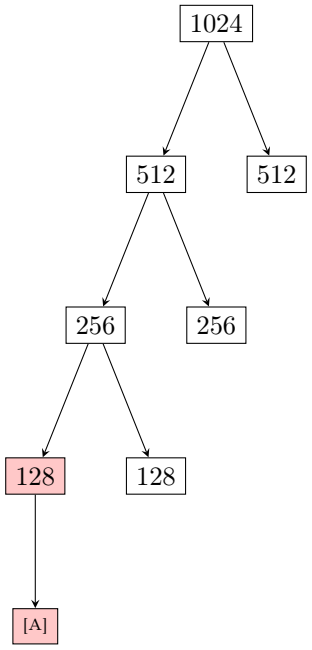



Figure 12: Buddy System Binary Tree

### 8.2 Virtual Memory System

Page table simulation with address translation:

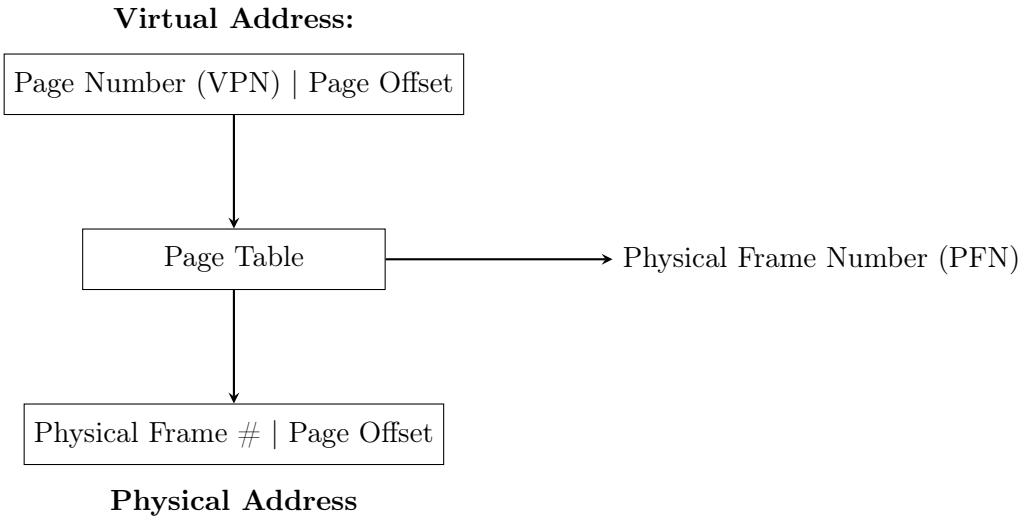



Figure 13: Virtual Memory Address Translation

### 8.3 Page Replacement Algorithms

- **FIFO**: Replace oldest page
- **LRU**: Replace least recently used
- **Clock (Second Chance)**: Circular list with reference bit
- **Optimal**: Replace page used furthest in future (theoretical)

### 8.4 TLB Simulation

Fast address translation cache:

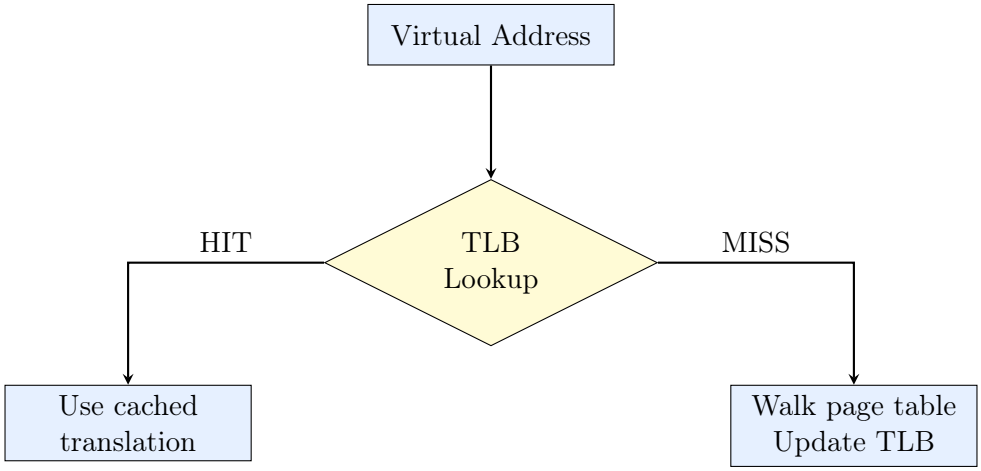



Figure 14: TLB Operation Flow

## Appendix A: Command Reference

| Command | Description | Example |
|---|---|---|
| `init memory <size>` | Initialize memory pool | `init memory 1024` |
| `set allocator <strategy>` | Set allocation algorithm | `set allocator best_fit` |
| `malloc <size>` | Allocate memory | `malloc 100` |
| `free <id>` | Free block by ID | `free 1` |
| `dump memory` | Show memory layout | `dump memory` |
| `stats` | Show allocation statistics | `stats` |
| `init cache` | Configure cache (interactive) | `init cache` |
| `cache read <addr>` | Simulate cache read | `cache read 0x1000` |
| `cache write <addr>` | Simulate cache write | `cache write 0x1000` |
| `cache stats` | Show cache statistics | `cache stats` |
| `cache config` | Show cache configuration | `cache config` |
| `cache reset` | Reset cache statistics | `cache reset` |

## Appendix B: Build Instructions

Build Commands

```
# Release build (optimized)
make

# Debug build (with symbols)
make debug

# Clean build artifacts
make clean

# Run the simulator
./build/memsim

# Run with workload file
./build/memsim < tests/workload1_basic.txt
```

## Appendix C: Test Workloads

The `tests/` directory contains workload files for testing:

| File | Purpose |
|---|---|
| `workload1_basic.txt` | Basic allocation/deallocation |
| `workload2_strategies.txt` | Compare allocation strategies |
| `workload3_cache.txt` | Cache hit/miss patterns |
| `workload4_stress.txt` | Stress testing |
| `workload5_edge_cases.txt` | Boundary conditions |
| `workload6_cache_policies.txt` | FIFO vs LRU comparison |

Table 10: Test Workload Files