Memory Management Simulator - Technical Documentation

Implementation Summary

✅ IMPLEMENTED FEATURES:

- **Physical Memory Management**: Linked-list based allocation with first-fit, best-fit, and worst-fit strategies
- **Cache Hierarchy**: Two-level (L1/L2) cache simulation with LRU replacement policy
- **Virtual Memory**: Page-based virtual memory with address translation and page fault handling
- **Interactive CLI**: Command-line interface for testing all subsystems
- **Memory Coalescing**: Automatic merging of adjacent free blocks

---

Table of Contents

Memory Layout and Assumptions

Physical Memory Structure

The simulator implements a **linked-list based memory management system** that maintains a continuous view of physical memory.

```
Physical Memory Layout:
┌─────────────────────────────────────────────────────────┐
│                    Total Memory Size                    │
├──────────────┬──────────────┬──────────────┬──────────────┤
│   Block 1    │   Block 2    │   Block 3    │   Block N    │
│ (Free/Used)  │ (Free/Used)  │ (Free/Used)  │ (Free/Used)  │
└──────────────┴──────────────┴──────────────┴──────────────┘

Each Block Contains:
- start_address: Starting address of the block
- size: Size of the block in bytes
- is_free: Boolean indicating if block is available
- id: Unique identifier for allocated blocks
- next: Pointer to next block in linked list
```

Key Assumptions

1. **Contiguous Memory**: Physical memory is treated as a single contiguous block
2. **Word-Aligned Access**: All memory addresses are word-aligned (4-byte boundaries)
3. **No Memory Protection**: No hardware-level memory protection mechanisms
4. **Simplified Addressing**: Linear addressing model without segmentation
5. **Static Memory Pool**: Total memory size is fixed at initialization
6. **No Fragmentation Handling**: Beyond basic coalescing of adjacent free blocks

Memory Block Structure

```
struct block {
    int start_address;  // Starting address of the block
    int size;          // Size in bytes
    bool is_free;      // Availability status
    int id;           // Unique identifier for allocated blocks
    block* next;      // Pointer to next block
};
```

Allocation Strategy Implementations

The simulator implements three classic memory allocation strategies:

1. First Fit Algorithm

Strategy: Allocate the first free block that is large enough.

```
Algorithm Flow:
1. Traverse the linked list from the beginning
2. Find the first free block with size >= requested_size
3. If block size > requested_size, split the block
4. Mark the allocated portion as used
5. Return the block ID

Time Complexity: O(n) where n is the number of blocks
Space Complexity: O(1)
```

Implementation Details:

- Fast allocation for small requests
- Can lead to external fragmentation at the beginning of memory
- Good locality of reference for sequential allocations

2. Best Fit Algorithm

Strategy: Allocate the smallest free block that can satisfy the request.

```
Algorithm Flow:
1. Traverse the entire linked list
2. Find all free blocks with size >= requested_size
3. Select the block with minimum size among candidates
4. If block size > requested_size, split the block
5. Mark the allocated portion as used
6. Return the block ID

Time Complexity: O(n) where n is the number of blocks
Space Complexity: O(1)
```

Implementation Details:

- Minimizes wasted space per allocation
- Requires full list traversal
- Can create many small unusable fragments

3. Worst Fit Algorithm

Strategy: Allocate the largest available free block.

```
Algorithm Flow:
1. Traverse the entire linked list
2. Find all free blocks with size >= requested_size
3. Select the block with maximum size among candidates
4. If block size > requested_size, split the block
5. Mark the allocated portion as used
6. Return the block ID

Time Complexity: O(n) where n is the number of blocks
Space Complexity: O(1)
```

Implementation Details:

- Leaves larger fragments after allocation
- May reduce external fragmentation
- Slower than first fit due to full traversal

Memory Coalescing

The simulator implements automatic coalescing of adjacent free blocks:

```
Coalescing Process:
1. When a block is freed, check adjacent blocks
2. If previous block is free, merge backwards
3. If next block is free, merge forwards
4. Update size and remove redundant block nodes
5. Maintain linked list integrity

Before Coalescing:
[Used][Free-50][Free-30][Used] → [Used][Free-80][Used]
```

Cache Hierarchy and Replacement Policy

Cache Architecture

The simulator implements a **two-level cache hierarchy** with configurable parameters:

```
Cache Hierarchy:
CPU → L1 Cache → L2 Cache → Main Memory
      (Fast)     (Medium)    (Slow)

L1 Cache: Small, fast, close to CPU
L2 Cache: Larger, slower, shared or private
```

Cache Structure

```
struct cacheline {
    int tag;        // Address tag for identification
    bool valid;     // Valid bit
    int data;       // Cached data (simplified)
    int timestamp;  // For LRU replacement
};

struct cachelevel {
    int size;           // Total cache size
    int block_size;     // Size of each cache line
    int associativity;  // Number of lines per set
    int num_sets;       // Number of cache sets
    vector<vector<cacheline>> cache; // 2D array of cache lines
    Memory* main_memory;  // Pointer to main memory
    int hits, misses;     // Performance counters
};
```

Address Mapping

```
Virtual Address Breakdown:
┌──────────────┬──────────────┬──────────────┐
│     Tag      │   Set Index  │   Offset     │
└──────────────┴──────────────┴──────────────┘

Set Index: Determines which cache set to use
Tag: Stored in cache line for hit/miss determination
Offset: Byte offset within the cache line (not used in simulation)
```

Cache Configuration Examples

```
Direct-Mapped Cache (Associativity = 1):
Set 0: [Line 0]
Set 1: [Line 1]
Set 2: [Line 2]
Set N: [Line N]

2-Way Set-Associative Cache (Associativity = 2):
Set 0: [Line 0][Line 1]
Set 1: [Line 2][Line 3]
Set N: [Line 2N][Line 2N+1]

Fully Associative Cache (Associativity = Total Lines):
Set 0: [Line 0][Line 1][Line 2]...[Line N]
```

LRU Replacement Policy

Least Recently Used (LRU) replacement is implemented using timestamps:

```
Algorithm:
1. On cache hit: Update timestamp to current time
2. On cache miss:
   - If empty line exists, use it
   - Else, find line with oldest timestamp
   - Replace oldest line with new data
   - Update timestamp

Implementation:
- Each cache line has a timestamp field
- Global counter increments on each access
- Replacement selects minimum timestamp line
```

Cache Performance Metrics

```
Hit Ratio Calculation:
hit_ratio = hits / (hits + misses)

Performance Counters:
- hits: Number of successful cache accesses
- misses: Number of cache misses requiring main memory access
- Total accesses: hits + misses
```

Virtual Memory Model

Virtual Memory Architecture

The simulator implements a **page-based virtual memory system** with address translation:

```
Virtual Memory Components:
┌──────────────────┐    ┌──────────────────┐    ┌──────────────────┐
│ Virtual Address  │ → │   Page Table     │ → │Physical Address  │
│     Space        │    │   Translation    │    │     Space        │
└──────────────────┘    └──────────────────┘    └──────────────────┘
```

Memory Organization

```
Virtual Address Space Layout:
┌─────────────────────────────────────────────────────┐
│                Virtual Memory Space                 │
│                  (e.g., 2048 bytes)                 │
├──────────────┬──────────────┬──────────────┬────────┤
│   Page 0     │   Page 1     │   Page 2     │   Page N    │
│ (256 bytes) │ (256 bytes) │ (256 bytes) │ (256 bytes) │
└──────────────┴──────────────┴──────────────┴────────┘

Physical Memory (Frames):
┌─────────────────────────────────────────────────────┐
│                Physical Memory Space                │
│                  (e.g., 1024 bytes)                 │
├──────────────┬──────────────┬──────────────┬────────┤
│   Frame 0    │   Frame 1    │   Frame 2    │   Frame M   │
│ (256 bytes) │ (256 bytes) │ (256 bytes) │ (256 bytes) │
└──────────────┴──────────────┴──────────────┴────────┘
```

Page Table Structure

```
struct PageTableEntry {
    bool valid;        // Page is loaded in physical memory
    int frame_number;  // Physical frame number if valid
    bool referenced;   // Recently accessed (for LRU)
    int timestamp;     // Last access time for replacement
};

struct frame {
    bool occupied;     // Frame is in use
    int page_number;   // Virtual page stored in this frame
    int timestamp;     // Last access time for LRU replacement
};
```

Virtual Memory Parameters

- **Virtual Memory Size**: Total virtual address space (e.g., 2048 bytes)
- **Physical Memory Size**: Available physical memory (e.g., 1024 bytes)
- **Page Size**: Size of each page/frame (e.g., 256 bytes)
- **Number of Virtual Pages**: virtual_size / page_size
- **Number of Physical Frames**: physical_size / page_size

Address Translation Flow

Translation Process

```
Address Translation Algorithm:
1. Extract page number from virtual address
   page_number = virtual_address / page_size

2. Extract offset within page
   offset = virtual_address % page_size

3. Check page table entry for the page
   entry = page_table[page_number]

4. If page is valid (in physical memory):
   physical_address = entry.frame_number * page_size + offset

5. If page is not valid (page fault):
   - Find free frame or evict a page using LRU
   - Load page into physical frame
   - Update page table entry
   - Compute physical address
```

Detailed Translation Steps

```
Step-by-Step Translation:
Virtual Address: 500 (example)
Page Size: 256 bytes

Step 1: Calculate page number
page_number = 500 / 256 = 1 (integer division)

Step 2: Calculate offset
offset = 500 % 256 = 244

Step 3: Lookup page table
entry = page_table[1]

Step 4a: If entry.valid == true
physical_address = entry.frame_number * 256 + 244

Step 4b: If entry.valid == false (Page Fault)
- Select victim frame using LRU policy
- Update victim's page table entry (mark invalid)
- Load page 1 into selected frame
- Update page_table[1] with frame number and valid=true
- Calculate physical_address = frame_number * 256 + 244
```

Page Fault Handling

```
Page Fault Resolution:
1. Page fault occurs when accessing invalid page
2. Find replacement frame using LRU policy
3. If replacement frame contains valid page:
   - Mark corresponding page table entry as invalid
4. Load new page into selected frame
5. Update page table entry for new page
6. Update timestamps for LRU policy
7. Complete address translation
```

LRU Page Replacement

```
LRU Frame Selection Algorithm:
1. Scan all physical frames
2. Find frame with minimum timestamp
3. Select frame for replacement
4. Update victim page's page table entry
5. Load new page and update timestamps

Implementation:
int oldest_timestamp = INT_MAX;
int lru_frame = -1;
for (int i = 0; i < num_frames; i++) {
    if (frames[i].timestamp < oldest_timestamp) {
        oldest_timestamp = frames[i].timestamp;
        lru_frame = i;
    }
}
```

Limitations and Simplifications

Memory Management Limitations

1. **No Memory Protection**
    - No hardware-level access controls
    - No user/kernel mode distinction
    - No memory segmentation or protection rings
2. **Simplified Address Space**
    - Linear addressing model only
    - No support for segmentation
    - Fixed memory pool size
3. **Basic Fragmentation Handling**
    - Only adjacent block coalescing
    - No compaction or defragmentation
    - No buddy system for optimal allocation
4. **Limited Error Handling**
    - Minimal bounds checking
    - No out-of-memory handling strategies
    - Basic input validation only

Cache System Limitations

1. **Simplified Cache Model**
    - No cache coherency protocols
    - No multi-core considerations
    - No write-back/write-through policies
2. **Basic Replacement Policy**
    - Only LRU replacement implemented
    - No FIFO, Random, or adaptive policies
    - Simple timestamp-based LRU
3. **No Cache Hierarchy Optimization**
    - No inclusive/exclusive cache policies
    - No prefetching mechanisms
    - No cache line state management
4. **Performance Modeling**
    - No actual timing simulation
    - No memory latency modeling
    - Simplified hit/miss counting

Virtual Memory Limitations

1. **Simplified Page Management**
    - No demand paging with disk storage

- No page aging algorithms
- Basic LRU replacement only

2. **No Advanced Features**
   - No copy-on-write pages
   - No shared memory pages
   - No memory-mapped files
3. **Limited Translation**
   - Single-level page table only
   - No translation lookaside buffer (TLB)
   - No multi-level page tables
4. **No Process Management**
   - Single address space only
   - No process isolation
   - No context switching

Implementation Simplifications

1. **Data Structures**
   - Simple linked lists instead of optimized structures
   - No balanced trees or hash tables
   - Basic array-based implementations
2. **Threading and Concurrency**
   - Single-threaded simulation only
   - No race condition handling
   - No atomic operations
3. **Hardware Simulation**
   - No actual hardware timing
   - Simplified instruction execution
   - No pipeline or superscalar modeling
4. **Memory Hierarchy**
   - No NUMA considerations
   - No memory controller simulation
   - Simplified bus and interconnect model

Educational vs. Production Considerations

This simulator is designed for **educational purposes** and includes several simplifications that would not be acceptable in a production memory management system:

- **Performance**: O(n) allocation algorithms instead of O(log n) optimized versions
- **Reliability**: Limited error handling and recovery mechanisms
- **Scalability**: Fixed-size data structures and simple algorithms
- **Compatibility**: No consideration for different architectures or standards

Future Enhancement Opportunities

1. **Advanced Allocation**: Implement buddy system, slab allocators, or segregated free lists
2. **Better Cache Models**: Add write policies, cache coherency, and multi-level optimization
3. **Enhanced Virtual Memory**: Add TLB simulation, multi-level page tables, and demand paging
4. **Performance Tools**: Add detailed performance analysis and visualization
5. **Multi-threading**: Support for concurrent access and thread-safe operations

---

Conclusion

This memory management simulator provides a solid foundation for understanding core memory management concepts including physical memory allocation, cache hierarchies, and virtual memory systems. While simplified for educational purposes, it demonstrates the fundamental algorithms and data structures used in real operating systems and computer architectures.

The modular design allows for future enhancements and provides a clear separation between different memory management subsystems, making it an effective tool for learning and experimentation with memory management techniques.