MEMORY MANAGEMENT SIMULATOR

Vijita Gabra

24116106

II Yr, Electronics & Communication

DESIGN DOCUMENT

OVERVIEW

Memory management is a core responsibility of an operating system, involving the allocation, deallocation, caching, and virtualization of memory resources.

This project implements a **memory management simulator** that models how an operating system manages **physical memory**, **CPU caches**, and **virtual memory**.

The simulator is implemented as a **user-space, command-line application** and focuses on **algorithmic correctness**, **system-level abstractions**, and **performance trade-offs**, rather than hardware control or kernel development.

OBJECTIVES

The objectives of this project are:

- To understand and implement **dynamic memory allocation strategies**
- To study **memory fragmentation** and its impact
- To simulate **CPU cache hierarchies and replacement policies**
- To model **virtual memory using paging**
- To gain hands-on experience with **OS memory abstractions**

SIMULATION ASSUMPTIONS

To keep the simulator focused and tractable, the following assumptions are made:

- Single-process system
- Paging-based virtual memory (no segmentation)
- Fixed page size
- FIFO page replacement
- Inclusive cache hierarchy
- No real disk I/O (symbolic memory access)
- User-space simulation via CLI

These assumptions are **explicit and consistent across all modules**.

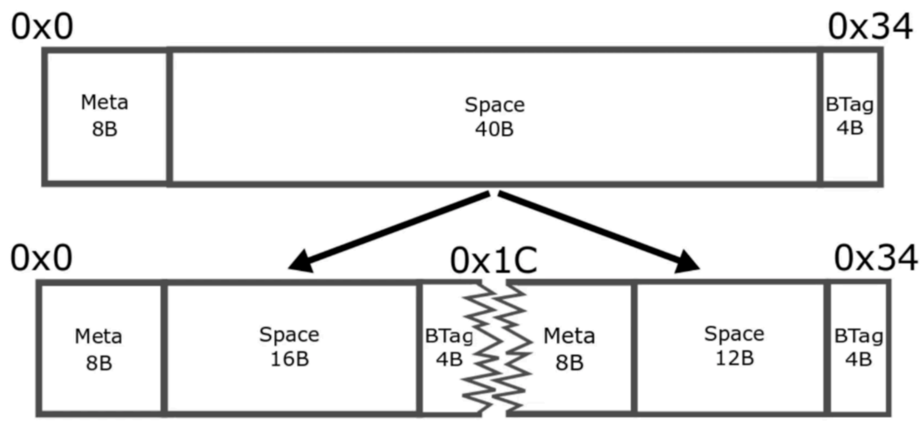
PHYSICAL MEMORY SIMULATION

Design

- Physical memory is simulated as a **contiguous block**
- Memory is dynamically divided into variable-sized blocks
- Blocks are represented using a linked list

Data Structure

```
struct Block {
    int start;
    int size;
    int requestedSize;
    bool free;
    int id;
};
```



Features Implemented

- Dynamic block splitting
- Explicit free/allocated tracking
- Address-based memory visualization

MEMORY ALLOCATION STRATEGIES

The simulator implements the following allocation algorithms:

- First Fit
- Best Fit
- Worst Fit

Allocation Process

1. Traverse free blocks based on selected strategy
2. Allocate memory if sufficient space exists
3. Split blocks when necessary
4. Assign a unique block ID

Deallocation

- Frees a block by ID
- **Coalesces adjacent free blocks** to reduce fragmentation

This avoids the common pitfall of unmerged free blocks.

FRAGMENTATION METRICS

Internal Fragmentation

Internal fragmentation is computed as:

$$AllocatedBlockSize - RequestedSize$$

- Tracked per allocated block
- Aggregated in statistics

In the variable-size allocator, internal fragmentation is often zero because allocation is exact.

However, internal fragmentation is demonstrated clearly in the **Buddy Allocator**.

External Fragmentation

External fragmentation is computed as:

$$1 - (LargestFreeBlock / TotalFreeMemory)$$

BUDDY ALLOCATION SYSTEM

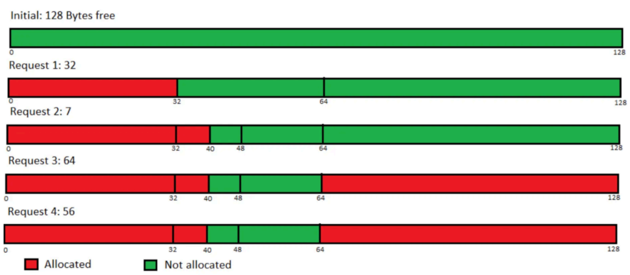
Design

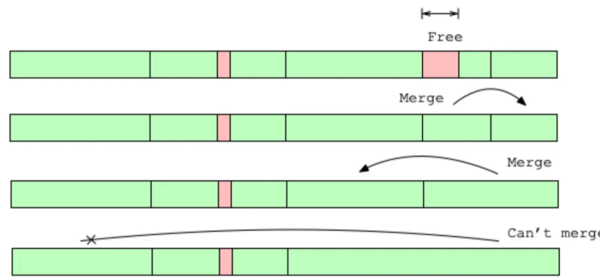
The buddy allocator enforces **power-of-two block sizes** to reduce external fragmentation.

Key Features

- Recursive block splitting
- XOR-based buddy address calculation
- Free lists indexed by block size
- Automatic buddy coalescing on free

This allocator demonstrates the trade-off between **external** and **internal fragmentation**.



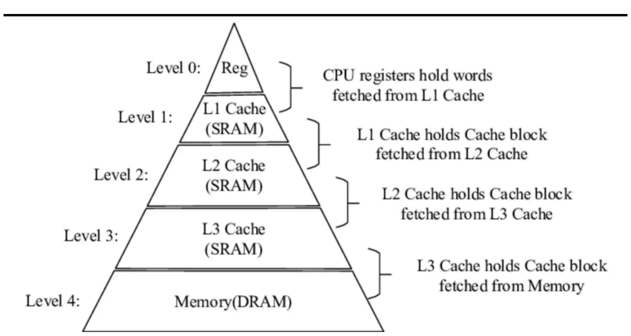


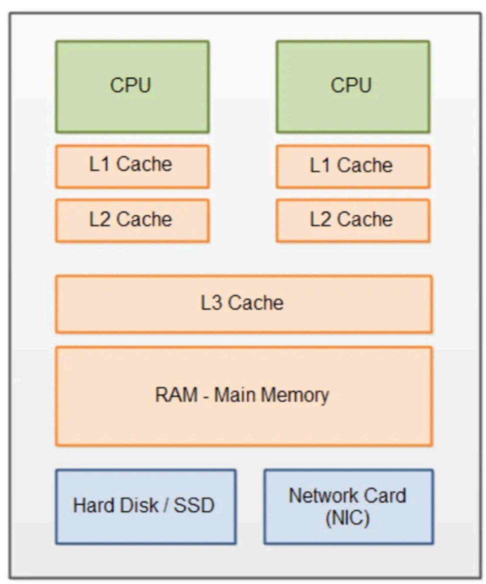
MULTILEVEL CACHE SYSTEM

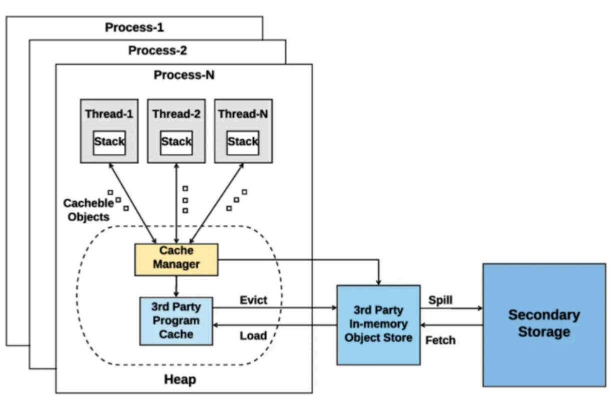
Cache Hierarchy

The simulator models a multilevel CPU cache:

> CPU → L1Cache → L2Cache → Main Memory







Configurable Parameters

- Cache size
- Block size
- Associativity (direct-mapped or N-way set associative)
- Replacement policy (LRU)

Replacement Policy

- **Least Recently Used (LRU)** replacement
- Implemented using timestamps
- Eviction occurs within a set for set-associative caches

Statistics

- Cache hits
- Cache misses
- Hit rate per cache level

Cache logic is **fully decoupled** from memory allocation logic.

VIRTUAL MEMORY SIMULATION

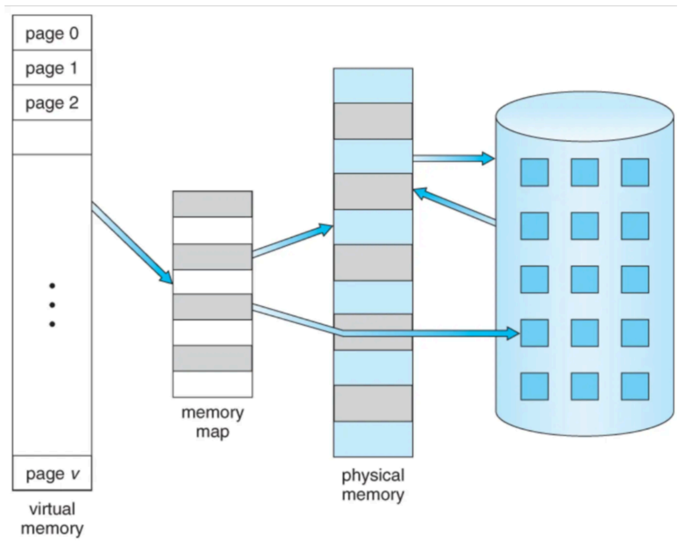
Virtual memory is implemented using paging.

Address Translation

> **VirtualAddress**
> →Page Number +Offset
> →PageTable
> → Frame Number
> → PhysicalAddress

Page Table Entry

```
structPageTableEntry {
    bool valid;
    int frameNumber;
};
```



Page Replacement

- FIFO replacement policy
- Replacement metadata maintained via a queue
- Page faults and hits are tracked explicitly

Statistics

- Page hits
- Page faults
- Hit rate

COMMAND LINE INTERFACE

The simulator provides an interactive CLI supporting:

Memory Allocation

```
init <size>
set <first|best|worst>
malloc <size>
free <id>
dump
stats
```

Buddy Allocator

```
init_buddy <total><min_block>
malloc<size>
free<address><size>
dump_buddy
```

Cache

```
init_cache <l1><l2><block><assoc>
access<address>
cache_stats
```

Virtual Memory

```
init_vm <virtual><physical><page_size>
vm_access<address>
vm_stats
```

Testing and Validation

The simulator was tested using:

- Sequential allocation/deallocation patterns
- Cache access patterns demonstrating locality and conflict misses
- Paging workloads causing page faults and replacements

Observed outputs matched expected theoretical behavior.

Limitations and Future Work

- LFU and Clock cache replacement policies
- LRU page replacement
- Multi-process virtual memory
- Disk access latency simulation
- Graphical visualization

Conclusion

This project successfully simulates core operating system memory management mechanisms, including allocation strategies, cache hierarchies, and virtual memory.

The modular design enables clear analysis of trade-offs between performance and fragmentation, making the simulator a strong educational and experimental tool.

TEST ARTIFACTS

Memory Allocation Workloads

Sequential Allocation and Deallocation

Purpose:

To verify correct allocation, deallocation, and coalescing of free blocks.

Input Workload:

```
init 256
set first
malloc 64
malloc 32
free 1
```

```
dump
stats
```

Expected Behavior / Correctness Criteria:

- Block `id=1` is freed
- Adjacent free blocks are merged
- Memory dump reflects correct block boundaries
- External fragmentation is reduced after coalescing

Validation Result:

- Free blocks are merged correctly
- No memory leaks or overlapping blocks

Allocation Strategy Comparison

Purpose:

To compare First Fit, Best Fit, and Worst Fit behavior.

Input Workload:

```
init 512
set best
malloc 100
malloc 200
free 1
malloc 80
dump
```

Expected Behavior:

- Best Fit chooses the smallest sufficient free block
- Memory layout differs from First Fit and Worst Fit

Correctness Criteria:

- Allocation location changes with strategy
- No allocation occurs in insufficient blocks

Validation Result:

- Strategy-specific behavior observed

Fragmentation Metrics Validation

External Fragmentation

Input Workload:

```
init 128
malloc 30
malloc 20
free 1
stats
```

Expected Output:

- External fragmentation computed as

> 1 – (largest_free_block / total_free_memory)

Validation Result:

- External fragmentation reported correctly

Internal Fragmentation

Input Workload (Buddy Allocator):

```
init_buddy 128 16
malloc 30
```

Expected Behavior:

- Request rounded to 32 bytes
- Internal fragmentation = 2 bytes

Correctness Criteria:

- Allocated block size > requested size
- Fragmentation explained by power-of-two rounding

Validation Result:

- Internal fragmentation demonstrated and explained

Cache Access Logs

Temporal Locality Test

Purpose:

To verify cache hits due to repeated access.

Input Workload:

```
init_cache 64 128 16 2
access 0
access 0
access 0
cache_stats
```

Expected Output:

```
L1 MISS → L2 MISS → Memory Access
L1 HIT
L1 HIT
```

Correctness Criteria:

- First access causes a miss
- Subsequent accesses cause hits
- Hit rate > 0%

Validation Result:

- Temporal locality correctly exploited

Conflict Miss Test

Purpose:

To demonstrate conflict misses in set-associative cache.

Input Workload:

```
init_cache 64 128 16 2
access 0
access 64
access 128
access 0
access 64
cache_stats
```

Expected Behavior:

- All addresses map to the same cache set
- Cache thrashing occurs
- Hit rate may be 0%

Correctness Criteria:

- Evictions occur due to limited associativity
- Behavior matches theoretical cache mapping

Validation Result:

- Conflict misses correctly observed

Virtual Memory Access Logs

Page Fault and Page Hit Test

Purpose:

To validate paging and FIFO replacement.

Input Workload:

```
init_vm 1024 256 64
vm_access 0
vm_access 64
vm_access 128
vm_access 0
vm_access 256
vm_stats
```

Observed Sample Output:

```
Physical address:0
Physical address:64
Physical address:128
Physical address:0
Physical address:192
Page Hits:1
Page Faults:4
Hit Rate:20%
```

Correctness Criteria:

- First-time page accesses cause page faults
- Re-access causes page hit
- Physical address = frame × page size + offset
- FIFO replacement occurs when frames are full

Validation Result:

- Paging and replacement logic correct

> All tests can be executed by redirecting input files to the simulator binary. For example:

```
chmod +x tests/run_all_tests.sh
```

```
./memsim < tests/vm_tests.txt
```

> Sample execution logs are provided in `docs/logs/`