Memory management simulator

Modern operating systems do not directly allocate memory linearly. Instead, they rely on a combination of memory allocation strategies, virtual memory, paging, cache hierarchies, and replacement algorithms to manage limited physical resources efficiently.
This project implements a Memory Management Simulator in C++ that mimics major OS-level memory mechanisms, including physical memory partitioning, First-Fit / Best-Fit / Worst-Fit allocation, Buddy allocation, Virtual Memory paging, and a multilevel cache system with L1/L2 and replacement policies.

The simulator enables interactive experimentation using a CLI, allowing users to initialize memory, switch allocators, allocate/free blocks, access virtual addresses, and view statistics.

Memory Layout & Assumptions

In this simulator, the physical memory is modeled as a single continuous linear address space, similar to how physical RAM appears to an operating system at the hardware abstraction level. Instead of interacting with real hardware, the simulator creates a software-based virtual representation of memory, allowing controlled allocation, deallocation, and fragmentation analysis.

1.1 Continuous Memory Representation

The entire memory space is treated as one contiguous block of bytes. The size of this block is configurable by the user (e.g., 1024 bytes, 2048 bytes). This block is divided dynamically based on allocation requests. No pre-defined partitions exist at initialization, the simulator creates or splits blocks only when memory is allocated.

Example conceptual layout:

```
0 ————————————————————————————————————— 1023
|           Continuous memory space     |
```

1.2 Block Metadata & Tracking

Instead of storing data, each allocation or free segment is represented through an internal Block metadata structure, which maintains:

- starting address of the block,
- total size of the block,
- whether the block is free or allocated,
- block ID to reference allocation,
- and requested size (for internal fragmentation calculation).

1.3 Dynamic Allocation Behavior

When a request is made (malloc size), the system scans free blocks and selects memory according to the chosen allocation strategy (First Fit, Best Fit, Worst Fit, or Buddy System).Once a suitable free region is found, the memory is split: part allocated, and the remainder (if any) becomes a new free block.On deallocation (free block_id), the freed block is marked as free and, in traditional allocators, adjacent free blocks are coalesced (merged) to reduce fragmentation.

1.4 Simplifying Assumptions

To keep the simulator focused on core OS concepts, some simplifications are made:

- There is no real data stored inside memory, only metadata tracking.
- Only one process is simulated.
- Fragmentation and merging operate logically, without time-based simulation.
- Alignment constraints are not enforced, requested sizes map directly to bytes.

Allocation Strategy Implementations

One of the core goals of this simulator is to demonstrate how different memory allocation strategies behave when managing a continuous pool of physical memory. These policies determine how the allocator chooses a free space for fulfilling a given memory request. Each strategy impacts speed, fragmentation, memory utilization, and overall system performance differently.

The simulator modularly implements four allocation strategies: First Fit, Best Fit, Worst Fit, and Buddy Allocation. Each allocator is derived from a common base interface (Allocator), enabling the user to switch between strategies at runtime via CLI.

2.1 First-Fit Allocation

First-Fit scans memory from the beginning and selects the first available free block that is large enough to satisfy the request.

Working Mechanism:

1.) Traverse the list of free blocks linearly
2.) Stop as soon as the first sufficient block is found.
3.) Allocate memory in that region.
4.) If excess space remains, split the block.

Advantages:

1.) Very fast — minimal search overhead
2.) Good performance for general workloads
3.) Easy to implement

Disadvantages:

1.) Tends to create fragmentation near the beginning of memory
2.) Over time, free space gets scattered

Example Scenario:

Free Blocks: [0–99] [100–499] [500–1023]
malloc 80 → allocated at [0–79]

2.2 Best-Fit Allocation

Best-Fit examines all free blocks and chooses the block that leaves the least leftover free space, i.e., *the smallest block that fits the request*.

Working Mechanism:

1.) Scan entire memory
2.) Track smallest block ≥ request size
3.) Allocate within that smallest hole
4.) Leave remaining space (if any) as a new free block

Advantages:

1. Reduces internal fragmentation
2. Tries to utilize space efficiently

Disadvantages:

1. Slower than First Fit (full scan required)
2. Can lead to many tiny unusable holes, increasing external fragmentation

Example:

Free Blocks: 200, 350, 90, 500

malloc 180 → Best-Fit selects block of 200 (not 350 or 500)

2.3 Worst-Fit Allocation

Worst-Fit chooses the largest available free block and allocates space from there.

Working Mechanism:

1.) Scan memory
2.) Identify largest free region
3.) Allocate requested bytes from it
4.) Leave leftover section as a (still large) free block

Advantages:

1. Reduces the chance that large continuous blocks get completely broken
2. Sometimes helps keep medium blocks available for future requests

Disadvantages:

1. Slowest among simple strategies due to full scan
2. Can still suffer fragmentation if requests vary a lot

Example:

Free: [0–99] (100), [100–499] (400), [500–1023] (524)

malloc 120 → allocate inside [500–1023] (largest space)

Strategy	Search Time	Fragmentation	Best Use Case
First-Fit	Fastest	Moderate external frag	General programs, unpredictable workloads
Best-Fit	Slowest (full scan)	Low internal, *high external*	When memory is scarce & request sizes vary slightly
Worst-Fit	Slowest	Sometimes lowers external frag	Systems with frequent large allocations
Buddy	Fast for alloc/free	Internal frag due to rounding	OS Kernels, real-time systems

Buddy Allocation System

The Buddy Memory Allocation algorithm is a dynamic memory management technique that divides memory into power-of-two sized blocks and allocates them in pairs called *buddies*. It was originally designed to provide a fast and efficient allocation–deallocation strategy, especially within operating systems, because it drastically simplifies memory coalescing and reduces external fragmentation.

In this simulator, Buddy Allocation represents the fourth and most advanced allocator, following First-Fit, Best-Fit, and Worst-Fit. It mimics real-world kernel allocators used in Linux kmalloc() and BSD memory subsystems.

3.1 Conceptual Working

The Buddy System starts with the entire memory represented as one large block. When a request comes in, the allocator:

1.) Rounds the requested size up to the nearest power of two
2.) Splits existing blocks recursively in half until a block of suitable size is reached
3.) Assigns that block to the requester

Every split produces two blocks of equal size, called "buddies."
This hierarchical splitting continues like a binary tree.

Example:

Initial Block = 1024 bytes

Request = 200 bytes → round to 256

1024 → split → 512 + 512

512 → split → 256 + 256

→ allocate 256 block

3.2 Deallocation & Buddy Merging

The key advantage of the Buddy System is **automatic coalescing**. When an allocated block is freed:

- Its buddy is computed mathematically (buddy = start_address XOR block_size)
- If the buddy is also free, they are merged into a larger block
- Merging continues recursively upward

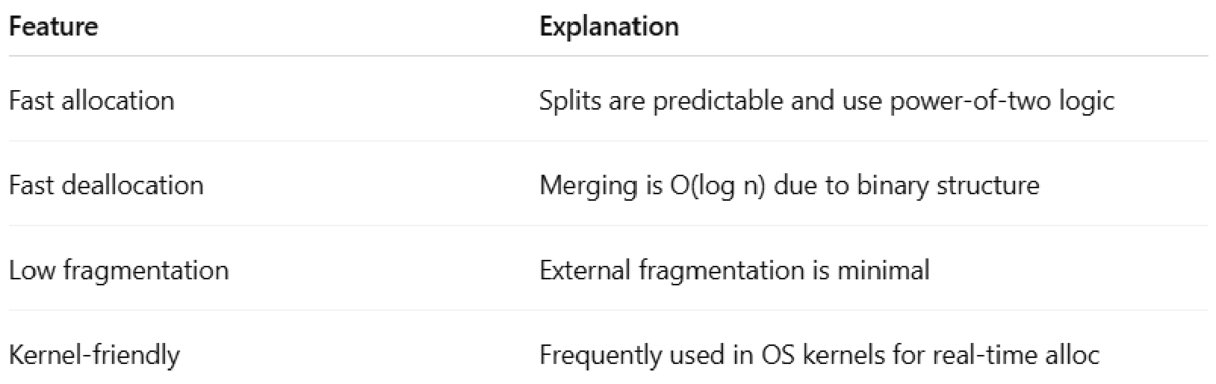
Example:

Free 256 @ 0

Buddy @ 256 exists and is free

→ merge → new free block @ 0 size 512

3.3 Advantages of the Buddy Method



Feature	Explanation
Fast allocation	Splits are predictable and use power-of-two logic
Fast deallocation	Merging is O(log n) due to binary structure
Low fragmentation	External fragmentation is minimal
Kernel-friendly	Frequently used in OS kernels for real-time alloc

Virtual Memory Model

Virtual Memory is a technique used by modern operating systems to provide a process with the *illusion* of having access to a large, continuous address space, even if the actual physical memory available is limited. Instead of forcing programs to live entirely within RAM, Virtual Memory allows only the required parts of a program to be loaded on-demand, while the remaining data can reside on secondary storage.

In this simulator, Virtual Memory is logically implemented (not actually interacting with a disk). The goal is to mimic OS paging behavior, page faults, and address translation.

4.1 Virtual–to–Physical Address Mapping

Every address generated by a program is a virtual address.
To access memory, this virtual address must be translated into a real physical frame address.

The simulator implements this translation using a Page Table:

Virtual Address → Page Number → Page Table Lookup → Physical Frame

4.2 Paging Parameters in the Simulator

- Virtual Memory Size → total virtual addressable bytes (e.g., 16 KB logically)
- Physical Memory Size → RAM being simulated (e.g., 1024 bytes)

- Page Size → fixed-size chunks (e.g., 64 bytes)
- Frames → physical memory pages = PhysicalMemory / PageSize

4.3 Page Table

The Page Table stores mappings of:

Virtual Page → Physical Frame

Each entry may contain:

- Frame number
- Valid/invalid bit
- Reference info (optional)

4.4 Integration With Cache System

Virtual Memory is not independent, it works together with CPU cache. Full access pipeline:

Virtual Address

↓

Page Table Translation

↓

Physical Frame Address

↓

L1 Cache Lookup

↓ (miss)

L2 Cache Lookup

↓ (miss)

Physical RAM Read

4.5 Page Fault Handling

A page fault happens when a virtual page requested by the user is not currently loaded into physical memory.

Simulator process:

1.) User accesses virtual address
2.) Page table checked
3.) If not present → page fault
4.) Replacement policy chooses a frame to evict

5.) Old page is removed (logically)
6.) New page is loaded & page table updated
7.) Statistics increment fault counter

Although the simulator does NOT simulate disk latency, it counts:

- page hits
- page faults

allowing performance evaluation.

Multilevel Cache Simulation

To properly model realistic operating system behavior, memory access must not be treated as a direct read from physical RAM. Modern CPUs use a multilevel cache hierarchy to significantly reduce access latency and improve performance.
This simulator includes a simplified but conceptually accurate model of a two-level cache consisting of L1 Cache (fastest, smallest) and L2 Cache (larger, slower).

The purpose of including caching is to demonstrate how temporal locality, spatial locality, and replacement policies affect the effective performance of memory access operations within a computer system.

5.1 Cache Levels

- **L1 Cache**
  - Closest to CPU
  - Smallest storage capacity
  - Fastest lookup time
  - High hit-rate for repeated accesses
- **L2 Cache**
  - Sits between L1 and physical memory
  - Larger capacity
  - Slightly slower
  - Acts as a second chance before falling back to RAM

5.2 Cache Organization

Caches are implemented using sets and cache lines, mimicking a simplified real CPU cache.

A cache line contains:

- tag (block identifier)
- valid bit (whether the line is storing useful data)
- last accessed timestamp or frequency counter (if using LRU or LFU)

A cache set is a container of multiple cache lines and represents associativity.

Example:

L1 Cache — 4 sets, each with 2 lines → 2-way associative

5.3 Cache Access Process

Every time the user performs a memory access (via CLI: access <address>), the simulator performs the following sequence:

1.) Map virtual address → frame → physical address
2.) Lookup in L1 cache
3.) If hit, return immediately
4.) If miss, lookup in L2 cache
5.) If miss again, retrieve from physical memory and insert into cache
6.) Replacement policy decides which line to evict, if cache is full

This is a simplified version of actual processor behavior, without timing simulation.

5.4 Replacement Policies

The simulator enables switching or configuring cache eviction policy. Supported policies include:

Policy	Meaning
FIFO	First-loaded block is evicted (queue-based)
LRU	Least-recently-used entry is removed (timestamp-based)
LFU (optional)	Least-frequently-used entry removed (counter-based)

5.5 Cache Statistics

To help analyze the effectiveness of caching, the simulator records:

- L1 cache hits
- L1 cache misses
- L2 cache hits
- L2 cache misses
- Hit ratio
- Miss propagation to lower levels

Address Translation Full Flow

The simulator models full OS behavior:

User runs → malloc(size)

↓

Allocator chooses physical region

↓

Virtual address access:

Virtual → Page Table → Frame → L1 Cache

↓ (on miss)

L2 Cache

↓ (on miss)

Physical Memory

This creates a complete architectural path similar to Linux paging + CPU cache.

Statistics & Reporting

Simulator prints stats:

Metric	Meaning
Used Memory	Sum of allocated blocks
Free Memory	Remaining space
Allocation success rate	successful allocations / total attempts
Utilization	used / total memory
Internal Fragmentation	allocated size – requested
External Fragmentation	largest free block ratio

Future Improvements

- Add multi-process address spaces
- Add TLB and multi-level paging
- Add GUI visualization
- Export execution logs as graphs