Memory Management Simulator - Design Document

1. Overview

* This project simulates a multiprogramming operating system's memory management unit.
* It includes a custom Physical Memory Allocator and a Two-Level CPU Cache hierarchy.
* The goal is to analyze fragmentation, memory utilization, and cache hit/miss behavior.

2. Architecture

The system is modularized into three main components:

* MemoryManager (src/allocator): Handles physical RAM storage, allocation strategies, and statistics.
* Cache (src/cache): Simulates L1/L2 cache logic, including hit/miss tracking and invalidation.
* Main (src/main.cpp): Provides the Command Line Interface (CLI) to interact with the system.

3. Memory Layout and Assumptions

* Physical Memory: Modeled as a contiguous logic space of bytes (simulated by a `total_size` variable).
* Block Structure: Memory is managed using a `Block` linked list. Each block contains metadata:

  * `id`: Unique identifier for the allocation (0 for free blocks).
  * `size`: Size of the block in bytes (including padding for alignment).
  * `start_addr`: Physical starting address.
  * `is_free`: Boolean flag indicating status.

4. Allocation Strategies

The allocator supports dynamic strategy switching at runtime:

* First Fit: Scans the list and selects the first block that satisfies the request.
* Best Fit: Scans the entire list to find the smallest block that fits (minimizes internal waste).
* Worst Fit: Scans the entire list to find the largest block (leaves big holes for future usage).

Coalescing:
When `free(id)` is called, the allocator scans the list. If the freed block has a free neighbor to the immediate left or right, they are merged into a single larger block to reduce external fragmentation.

5. Cache Hierarchy

The system simulates a two-level inclusive cache hierarchy:

* L1 Cache: Small, fast, Direct Mapped.

* L2 Cache: Larger, slightly slower, Direct Mapped.

* Flow of Execution:

  1. CPU requests Physical Address P.
  2. Check L1. If Hit -> Return immediately.
  3. If Miss -> Check L2.
  4. If L2 Hit -> Load block into L1 (Promote) -> Return.
  5. If L2 Miss -> Fetch from Main Memory -> Load into L2 -> Load into L1.

* Coherence Policy:
  When memory is freed in RAM, the simulator calculates which cache lines correspond to that address range and invalidates them in both L1 and L2 to prevent stale data access.

6. Usage (Build & Run)

* Compile (Manual):
```
g++ -o memsim src/main.cpp src/allocator/memory_manager.cpp src/cache/cache.cpp -I./include
```
* Compile (Make):
```
make
```
* Run:
```
./memsim
```

7. CLI Command Reference

The simulator accepts commands via standard input.

Command	Arguments	Description
init	<size>	Initialize RAM with <size> bytes. Clears all previous data.
config	<lvl> <size> <blk>	Configure Cache. lvl=1(L1) or 2(L2). <size> is total bytes, <blk> is block size.
strategy	<1-3>	Set Allocator: 1=First Fit, 2=Best Fit, 3=Worst Fit.
malloc	<size>	Allocate <size> bytes. Returns Block ID.
free	<id>	Free the memory associated with Block ID.
access	<addr>	Simulate CPU reading <addr>. Updates Cache L1/L2.
dump	(none)	Print table of all Cache contents and RAM Map (Free/Used blocks).
stats	(none)	Show Fragmentation %, Utilization %, and Cache Hit Ratios.
exit	(none)	Terminate the simulator.

8. Limitations

* Virtual Memory (Paging) is not included in this phase which may increase external fragmentation.
* The simulation is single-threaded; it does not simulate race conditions.
* Cache replacement is strictly Direct Mapped (conflict misses overwrite old data immediately).