MEMORY MANAGEMENT SIMULATOR

The project is a simulator of how the memory management is carried out by an operating system . It involved two different memory allocation modules namely – Variable Memory Division and Buddy Allocator System and connects them via L1 , L2 caching implementing the FIFO mechanism . The simulator also measures memory fragmentation , cache hits to misses etc.

MEMORY LAYOUT :

The complete memory size is initialised by the user by the ‘init’ command , from whereone the memory addresses goes from 0 to size-1. It uses objects to represent address ranges and thus simulate memory allocation . The simulator works in two different modes . The variable memory division uses linked list and the buddy allocator uses binary tree for the implementation. Then a multilevel set associative cache is also implemented for better performance.

ASSUMPTIONS :

Each process happens individually and asynchronously at a point of time. We do not store real data in memory. We only track information like block size or addresses in use etc. The memory is assumed to be one long line of memory locations starting from address 0 . Once the memory block is initialised , its size cannot be changed on the fly . In the variable memory division , free blocks are split to exactly the requested size , therefore leading to no internal fragmentation. In this mode memory blocks are referred to by ID instead of raw memory address. In the buddy allocator , the division is on the basis of next greater than or equal power of 2 , thus if the requested size is not a power of 2 , this leads to internal fragmentation . A block of size “x” must start at an address that is a multiple of “x” . In the caching system , we only handle hits and misses and not any CPU cycles or any real time data.

ALLOCATION STRATEGY AND IMPLEMENTATION :

The standard allocation strategy uses variable division . The implementation is done using doubly linked list (list<Block>) . Memory Blocks are frequently split and merged and thus a doubly linked list allows to keep memory blocks in physical order , fast insertion , simple access to neighboring blocks for merging . The simulator divides memory into blocks. Block is defined as a struct . It has the following fields namely “start” which is the starting address of the block , “size” stores the total size of the block , “requested_size” which stores the exact size the user asked for , “free” which indicates if the block is free or used , “id” which is a unique number given when a block is allocated which is used so the user can free memory without using the raw addresses .(see figure 1 for reference)

To decide where to place a memory request, the simulator scans the list of memory blocks. It supports three different allocation strategies, each with a different trade-off between speed and space usage (implemented in the findBlock function)

First Fit: In the First Fit strategy, the simulator starts scanning from the beginning of the block list and selects the first free block that is large enough to satisfy the request. The search stops as soon as a suitable block is found. This makes the strategy fast, but it can waste memory space by leaving poorly sized gaps.

Best Fit: In the Best Fit strategy, the simulator scans the entire list of blocks. It chooses the smallest free block that is still large enough for the request. By minimizing the leftover space, this strategy reduces wasted memory and preserves larger free blocks for future large allocations.

Worst Fit : In the Worst Fit strategy, the simulator also scans the entire list. It selects the largest free block available. The idea is to leave behind a large remaining free block, which is more likely to be useful later, instead of creating many small, unusable fragments.

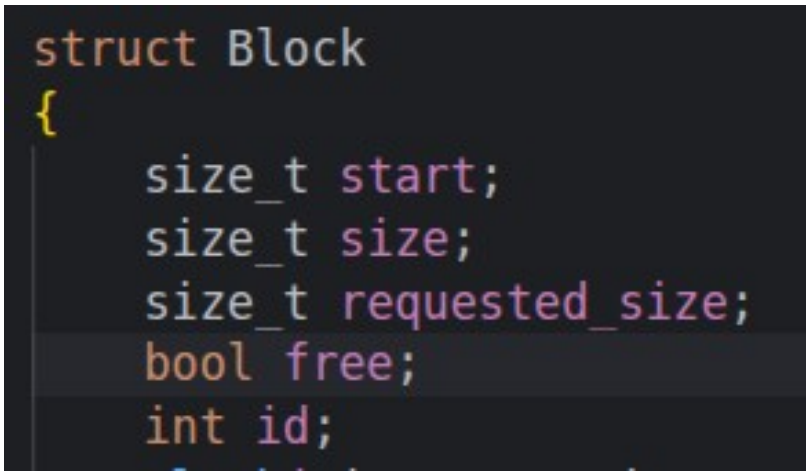

Figure 1: BLOCK STRUCT

After finding a valid block via any of the above strategies , the simulator begins the allocation process. (implemented in the mallocMem and splitBlock function) .
If the block is larger than the requested size, it is split into two parts. The first part becomes the allocated block and is sized exactly to the user's request. The remaining portion becomes a new free block that represents the unused space. This new free block is inserted immediately after the allocated block in the memory list, preserving the physical order of memory. After splitting, the allocated block is updated. It is marked as used, assigned a unique ID so it can be freed later, and its requested size is stored. This information is later used to calculate fragmentation statistics.

Freeing memory in the simulator happens through the functions freeMem and coalesce. First, the simulator searches the list of blocks to find the block that matches the given

ID. This ID uniquely identifies the allocated block that needs to be released. Once the block is found, it is marked as free and the merging process starts immediately. The simulator scans the list of blocks and checks neighboring blocks. If two adjacent blocks are both free, their sizes are added together to form a single larger block, and one of the blocks is removed from the list. By merging free blocks as soon as possible, the simulator keeps free memory in large, continuous chunks.

(see figure 2 for the complete flow of allocation and deallocation)

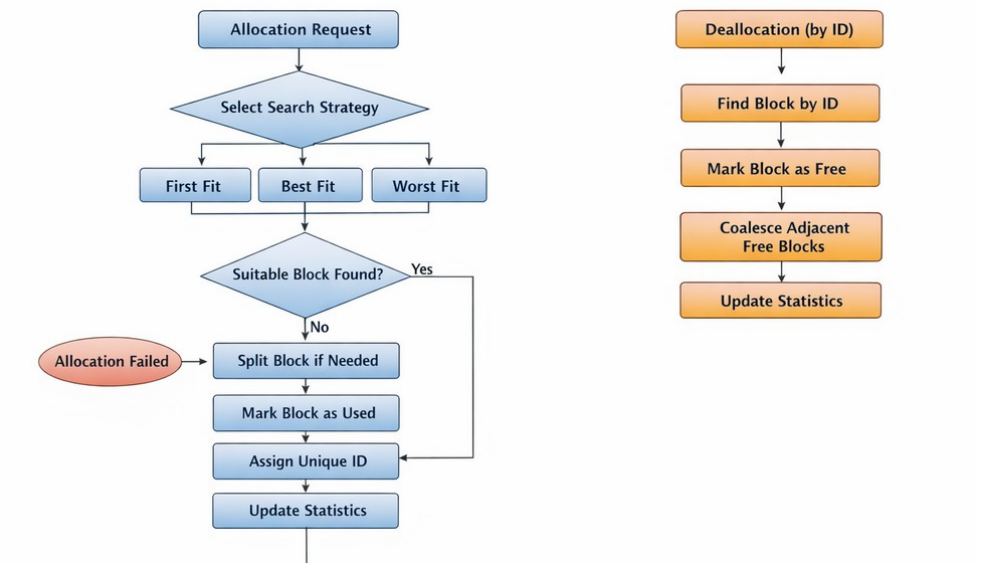

Figure 2: FLOWCHART OF ALLOCATION

BUDDY ALLOCATOR DESIGN:

The Buddy Allocator manages memory using power-of-two blocks (2^n). Unlike variable partitioning, this strict sizing allows very fast merging of free blocks, at the cost of internal fragmentation. The system is initialized with a total memory size and a minimum block size (both powers of two), which together define the maximum tree depth levelmax.
The free memory is tracked using std::vector<std::set<size_t>> freelist

Each index represents a level in the buddy tree. Level 0 holds the entire memory. Each next level contains blocks half the size of the previous one. std::set keeps free block addresses sorted with fast insertion and removal. User-requested sizes are tracked using std::map<size_t, size_t> blocks allocated . This maps a block's starting address to the original request size, allowing exact internal fragmentation calculation.

Allocation process is carried out as follows in the alloc function as follows :

1. The requested size is rounded up to the nearest power of two.

2. The allocator checks the free list at the required level.
3. If no block is available, it finds a larger block at a higher level.
4. The larger block is split repeatedly into two equal buddies until the correct level is reached.
5. Unused buddies are placed into their respective free lists.

The user always receives the smallest possible power-of-two block.

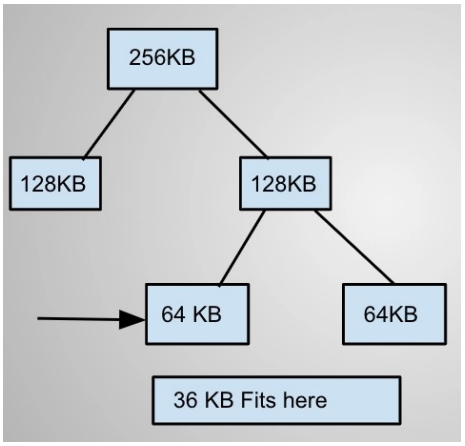

Figure 3: BUDDY ALLOCATION BINARY TREE

For the deallocation (freeBlock) , when a block is freed, its buddy is found using the expression:

BuddyAddress = Address XOR BlockSize

Buddy blocks differ by exactly one bit in their address. XOR with the block size flips this bit, giving the buddy address in constant time without scanning memory.

If the buddy is free, the two blocks are merged. This process repeats upward through the levels until merging is no longer possible or the top level is reached. This greedy coalescing tries to keep memory as contiguous as possible and minimizes external fragmentation.

The allocator also tracks totalused which is actual power-of-two memory allocated
totalrequest : memory requested by users

From these, it computes:

Internal Fragmentation = totalused − totalrequest
External Fragmentation = largest free block compared to total free memory
Utilization = percentage of memory serving user requests

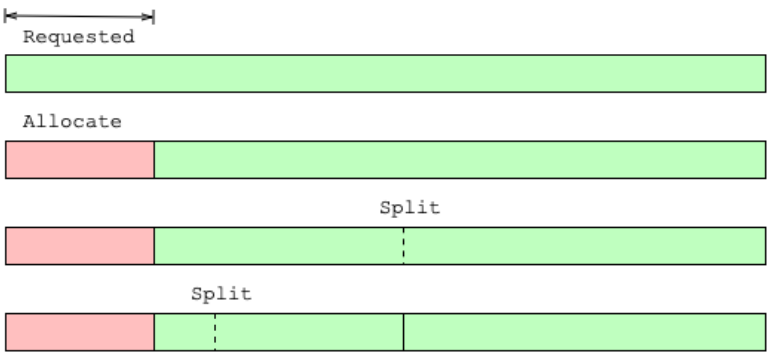

Figure 4: BUDDY ALLOCATION SPLIT

CACHE HIERARCHY

The simulator includes a multilevel cache module that models the fast memory between the CPU and physical RAM. It supports multiple cache levels such as L1 and L2, and forwards memory requests to the next level whenever a cache miss occurs. The cache follows a set-associative design, similar to real CPU caches.

The cache is implemented using a std::vector of std::deque, where each deque represents one cache set. The number of sets is calculated as the cache size divided by the product of block size and associativity. When associativity is 1, the cache behaves as a direct-mapped cache. When associativity is greater than 1, it works as an N-way set-associative cache.

For every memory access, the address is handled in the following steps:

1. The index is calculated to select the correct cache set.
2. The tag is calculated to identify the memory block within that set.

During a cache access, the simulator performs these steps:

1. It searches only within the selected set.
2. It checks each cache line for a valid bit and a matching tag.
3. If a match is found, the access is counted as a cache hit.

The FIFO replacement policy operates in these steps:

1. New cache lines are added at the back of the deque.
2. If the set is full during a miss, the oldest cache line at the front is removed.

Cache levels are connected in a simple hierarchy. An access is first checked in L1, then in L2 if needed, and finally treated as a main memory access if both levels miss. Each cache level tracks its own hits and misses to measure performance.

The simulator accurately models **Conflict Misses** when configured as a Direct-Mapped cache (associativity = 1), where multiple physical addresses map to the same set index. If two distinct tags compete for the same slot, the system performs a **FIFO eviction**, removing the existing block to make room for the new one. This behavior proves the mathematical correctness of our indexing logic and demonstrates the performance trade-offs inherent in low-associativity cache designs. (demonstration for the same is done in the Test Artifacts and the demonstration video)

To keep the simulator stable, a safety check ensures that invalid cache parameters do not cause errors. If the cache configuration is invalid, the number of sets is forced to one to prevent crashes.

The simulator uses a 1:1 implementation between virtual and physical address spaces. In this model, every address generated by a process is treated as a direct offset into the simulated physical memory. By using a 1:1

mapping, the simulator removes the complexity of page tables to provide a "pure" analysis of how the Buddy System and Standard Allocators manage the underlying physical blocks.

ADDRESS TRANSLATION FLOW

In this simulator, a hybrid translation model is used to support both ID-based memory management and direct physical address simulation. The translation path depends on which allocator or module is active, and each path follows a clear and well-defined flow.

When the Standard Memory Manager is used, the user does not work with physical addresses directly. Instead, the user interacts with Block IDs. During a malloc request, the user asks for N bytes. The system finds a suitable free block, splits it if needed, and assigns a unique ID such as ID 1. This ID is stored inside a Block structure within a std::list, forming the metadata that links the logical ID to a physical memory range.

When the user later requests to free a block using an ID, the system must translate that logical ID into a physical location. This translation is performed internally by scanning the std::list until a block with a matching ID is found. The process involves:

- Taking the input ID provided by the user
- Looping through the list until block.id equals the input ID
- Retrieving the physical start address (block.start) and the block size

Once this translation is complete, the physical address range from start to start plus size is marked as free, and adjacent free blocks are merged through coalescing.

When the Buddy Allocator or the Cache Simulation is used, the user works directly with hexadecimal physical addresses. In these modules, the simulator uses a 1:1 identity mapping, meaning the user-provided address is treated exactly as the physical address, with no virtual memory or paging layer involved.

For cache operations, this physical address is further processed using bit-slicing to locate its position in the cache. The steps include:

- Calculating the index as (address divided by block size) modulo the number of sets, which selects the target cache set
- Calculating the tag as (address divided by block size) divided by the number of sets, which identifies the memory block within that set

For buddy memory operations, such as free , the physical address is translated by looking it up in the blocksallocated map. This allows the system to identify the exact allocated block and perform the correct deallocation and merging operations.

SIMPLIFICATIONS:

- Omission of the paging and virtual memory layer : The simulator does not implement page tables, page directories, or a TLB. A 1:1 mapping is used where the virtual address is the same as the physical address. This removes address-translation overhead and allows direct analysis of Buddy System fragmentation and cache behavior.
- Single-threaded execution model : The simulator assumes a single-threaded environment and does not include mutexes, locks, or other concurrency controls. This keeps the focus on the correctness of allocation algorithms such as First Fit and Buddy splitting.
- Static memory and cache boundaries : The total memory size and cache configuration are fixed during initialization and cannot be changed at runtime. This reflects real hardware constraints where physical memory and caches have fixed sizes.

LIMITATIONS :

- No concurrent memory access : Because the system is single-threaded, it does not model race conditions, synchronization overhead, or parallel memory allocation behavior.
- Fixed cache replacement policy : Only FIFO replacement is implemented. Other policies such as LRU or LFU are not supported, which limits cache performance comparisons.