Memory Allocator

OpenAI Interview Questions

24 questions   Last Reviewed: Nov 24, 2025

#1 Interview Guide FREE PREVIEW

OpenAI Interview Guide

#2 Coding Software Engineer, Machine Learning Engineer 2025/11 FREE PREVIEW

Coding - Toy Language Type System

#3 Coding Software Engineer 2025/11

Coding - In-Memory Database with SQL Operations

#4 Coding Software Engineer 2025/10 Title Hidden

Problem Overview

Design and implement a memory allocator manager that manages a contiguous block of memory. Your implementation should support dynamic memory allocation and deallocation operations similar to `malloc()` and `free()` in C.

The allocator must:

- Initialize with a fixed total memory capacity
- Support allocating contiguous memory blocks
- Support freeing allocated memory blocks
- Handle memory fragmentation
- Raise errors for illegal operations or insufficient memory

Requirements

Implement a `MemoryAllocator` class with the following interface: [Source: darkinterview.com]

Constructor

```
# Source: https://darkinterview.com/collections/x7k9m2p4/questions/62d12301-1d6a-4ae8-8624-fc201f2804b0
def __init__(self, total_capacity: int)
```

- `total_capacity`: The total size of memory available for allocation (in bytes or abstract units)
- Initializes the allocator with all memory marked as free

Methods

1. `allocate(size: int) -> int`

Allocates a contiguous block of memory of the requested size.

Parameters:

`size`: The size of memory to allocate (must be positive)

Returns: [Source: darkinterview.com]

The starting address (index) of the allocated memory block

Behavior:

- Finds the first available free block that can accommodate the requested size (first-fit strategy)
- Marks the memory as allocated
- Returns the starting address of the allocated block

Errors:

- Raises an exception if `size` is invalid (non-positive)
- Raises an exception if there is insufficient contiguous memory available

2. `free(address: int, size: int) -> None`

Frees a previously allocated memory block. [Source: darkinterview.com]

Parameters:

- `address`: The starting address of the memory block to free
- `size`: The size of the memory block to free

Behavior:

- Marks the specified memory range as free
- Merges adjacent free blocks to reduce fragmentation

Errors: [Source: darkinterview.com]

- Raises an exception if the address is invalid (out of bounds)
- Raises an exception if attempting to free unallocated memory
- Raises an exception if the size is invalid

Part 1: Basic Implementation

Implement the memory allocator using a **linked list** data structure to track free memory blocks.

Data Structure

Use a doubly-linked list where each node represents a free memory block with:

- `start`: Starting address of the free block
- `size`: Size of the free block
- `next`: Pointer to the next free block
- `prev`: Pointer to the previous free block

Important: The free list should be maintained in address order (sorted by starting address) to enable efficient merging of adjacent blocks during deallocation. [Source: darkinterview.com]

Allocation Strategy

- **First-fit**: Traverse the free list from left to right and use the first block that can satisfy the allocation request
- If the free block is exactly the requested size, remove the node from the list
- If the free block is larger than requested, update the node's size and starting address

Deallocation Strategy

When freeing memory, handle the following cases:

1. **No merge**: The freed block is isolated (not adjacent to any free blocks)
   Insert a new node into the free list
2. **Merge with previous block**: The freed block is adjacent to the previous free block
   Extend the previous block's size
3. **Merge with next block**: The freed block is adjacent to the next free block
   Extend the freed block to include the next block and remove the next node
4. **Merge with both**: The freed block is between two free blocks
   Merge all three blocks into one

Example

```
# Source: https://darkinterview.com/collections/x7k9m2p4/questions/62d12301-1d6a-4ae8-8624-fc201f2804b0
# Initialize allocator with 100 units of memory
allocator = MemoryAllocator(100)

# Allocate 20 units
addr1 = allocator.allocate(20)  # Returns 0
# Memory: [Allocated(0-19)] [Free(20-99)]

# Allocate 30 units
addr2 = allocator.allocate(30)  # Returns 20
# Memory: [Allocated(0-19)] [Allocated(20-49)] [Free(50-99)]

# Allocate 40 units
addr3 = allocator.allocate(40)  # Returns 50
# Memory: [Allocated(0-19)] [Allocated(20-49)] [Allocated(50-89)] [Free(90-99)]

# Free the middle block
allocator.free(20, 30)
# Memory: [Allocated(0-19)] [Free(20-49)] [Allocated(50-89)] [Free(90-99)]

# Allocate 25 units (uses the freed block)
addr4 = allocator.allocate(25)  # Returns 20
# Memory: [Allocated(0-19)] [Allocated(20-44)] [Free(45-49)] [Allocated(50-89)] [Free(90-99)]

# Free first block
allocator.free(0, 20)
# Memory: [Free(0-19)] [Allocated(20-44)] [Free(45-49)] [Allocated(50-89)] [Free(90-99)]

# Free second block (merges with both adjacent free blocks)
allocator.free(20, 25)
# Memory: [Free(0-49)] [Allocated(50-89)] [Free(90-99)]
```

Part 2: Complexity Analysis and Optimization Discussion

After implementing the basic solution, be prepared to discuss:

Time Complexity

- **Allocation**: O(n) where n is the number of free blocks
  Must traverse the free list to find a suitable block
- **Deallocation**: O(n) in the worst case
  May need to traverse to find adjacent blocks for merging

Space Complexity

- O(m) where m is the number of free blocks
- In the worst case (maximum fragmentation with alternating allocated and free blocks), the number of free blocks can be significant
- The space complexity depends on the allocation/deallocation pattern rather than the total capacity

Drawbacks and Limitations

1. **External Fragmentation**: Over time, memory becomes fragmented with many small free blocks [Source: darkinterview.com]

    - Large allocation requests may fail even if total free memory is sufficient
    - Example: 100 units total, 60 free, but split into 6 blocks of 10 units each → cannot allocate 50 units
2. **Linear Search Time**: First-fit strategy requires O(n) time for allocation

3. **No Compaction**: Cannot relocate allocated blocks to reduce fragmentation

Optimization Strategies

Reducing Fragmentation: [Source: darkinterview.com]

1. **Segregated Free Lists**: Maintain separate free lists for different size classes

    - Small allocations (< 32 bytes) use a dedicated pool
    - Reduces fragmentation of the main memory space
    - Better cache locality
2. **Best-Fit Strategy**: Instead of first-fit, find the smallest free block that can satisfy the request

    - Reduces wasted space in each allocation
    - Trade-off: Slower allocation (still O(n) but examines all blocks)
3. **Buddy System**: Split memory into power-of-2 sized blocks [Source: darkinterview.com]

    - Simplifies merging (buddies are at predictable addresses)
    - Reduces fragmentation compared to arbitrary sizes
    - Internal fragmentation increases

Improving Time Complexity:

1. **Balanced Binary Search Tree (BST)**: Store free blocks in a BST with dual indexing

    - Maintain one tree ordered by **size** (for fast best-fit allocation in O(log n))
    - Maintain another tree ordered by **address** (for fast adjacent block lookup during merging in O(log n))
    - Alternatively, use a single tree with secondary index or augmented nodes
    - Total: O(log n) for both allocation and deallocation
    - Implementation options: Red-Black Tree, AVL Tree, or other self-balancing BST variants
    - Commonly used in production allocators (e.g., glibc's `ptmalloc2` uses Red-Black Trees)
2. **Bitmap + Index**: For fixed-size block allocation [Source: darkinterview.com]

    - Constant-time O(1) allocation with bit manipulation
    - Requires blocks to be uniform size

Achieving Constant Space Complexity:

To achieve O(1) space complexity:

1. **Implicit Free List**: Store metadata within the memory blocks themselves [Source: darkinterview.com]

    - Store block size and allocation status in a header at the start of each block (both free and allocated)
    - Free blocks can additionally store next/prev pointers in their payload area
    - No separate data structure needed - metadata lives in the managed memory itself
    - This is how most real-world allocators (including malloc) work
    - Minimum block size must accommodate the header and pointers (typically 16-24 bytes)
2. **Boundary Tags (Footer optimization)**: Used in conjunction with implicit free lists

    - Store size information at both the start (header) and end (footer) of each block
    - The footer enables O(1) backward traversal to find the previous block without a separate pointer
    - Enables O(1) coalescing with previous adjacent blocks (forward coalescing is already O(1))
    - Requires 2 * sizeof(size_t) overhead per block (header + footer)
    - This is a refinement of the implicit free list approach, not a separate technique

Follow-up Questions

Be prepared to answer:

1. How would you handle alignment requirements (e.g., all allocations must be 8-byte aligned)?
2. How would you track which memory blocks are currently allocated to detect invalid `free()` calls?
3. What would change if you needed to support `realloc()` (resize an allocated block)?
4. How would you implement a thread-safe version of this allocator?
5. What differences would arise in implementing this for actual hardware vs. as an abstract data structure?

Implementation Tips

Edge Cases to Handle

- Allocating size 0 or negative size
- Freeing memory at an invalid address
- Freeing memory that is already free (double-free)
- Freeing memory with an incorrect size
- Attempting to allocate more memory than total capacity
- Attempting to allocate when memory is fully fragmented

Testing Strategy

Create test cases for: [Source: darkinterview.com]

1. Basic allocation and deallocation
2. Memory exhaustion
3. Fragmentation scenarios
4. Merging free blocks in all combinations (prev, next, both)
5. Edge cases with allocations at boundaries (address 0, end of memory)
6. Invalid operations (double-free, out-of-bounds, etc.)

Sample Solution Framework

```
# Source: https://darkinterview.com/collections/x7k9m2p4/questions/62d12301-1d6a-4ae8-
8624-fc201f2804b0
class FreeBlock:
    def __init__(self, start: int, size: int):
        self.start = start
        self.size = size
        self.next = None
        self.prev = None


class MemoryAllocator:
    def __init__(self, total_capacity: int):
        if total_capacity <= 0:
            raise ValueError("Total capacity must be positive")

        self.capacity = total_capacity

        # Initialize with one large free block
        self.free_list_head = FreeBlock(0, total_capacity)

        # NOTE: This dictionary is for validation/debugging purposes only
        # A space-optimized version would use implicit free lists (storing
        # metadata within the memory blocks themselves) instead of this separate structure
        self.allocated = {}  # {address: size}

    def allocate(self, size: int) -> int:
        if size <= 0:
            raise ValueError("Allocation size must be positive")

        # Find first free block that fits (first-fit strategy)
        # Note: Free list is maintained in address order
        current = self.free_list_head
        while current:
            if current.size >= size:
                # Found suitable block
                allocated_address = current.start

                if current.size == size:
                    # Remove this block from free list
                    self._remove_free_block(current)
                else:
                    # Shrink the free block
                    current.start += size
                    current.size -= size

                # Track allocation
                self.allocated[allocated_address] = size

                return allocated_address
```

```
            current = current.next

        # No suitable block found
        raise MemoryError(f"Cannot allocate {size} units: insufficient contiguous memory")

    def free(self, address: int, size: int) -> None:
        if address < 0 or address >= self.capacity:
            raise ValueError(f"Invalid address: {address}")

        if size <= 0:
            raise ValueError("Size must be positive")

        if address + size > self.capacity:
            raise ValueError(f"Free range exceeds memory bounds")

        # Validate this was actually allocated
        if address not in self.allocated or self.allocated[address] != size:
            raise ValueError(f"Invalid free: no allocation at address {address} with size
{size}")

        # Remove from allocated tracking
        del self.allocated[address]

        # Find where to insert this free block
        # Check for merging with adjacent free blocks
        freed_end = address + size

        current = self.free_list_head
        prev_block = None

        # Find the position in the free list
        while current and current.start < address:
            prev_block = current
            current = current.next

        # Check merging scenarios
        can_merge_with_prev = prev_block and (prev_block.start + prev_block.size ==
address)
        can_merge_with_next = current and (freed_end == current.start)

        if can_merge_with_prev and can_merge_with_next:
            # Merge with both
            prev_block.size += size + current.size
            self._remove_free_block(current)
        elif can_merge_with_prev:
            # Merge with previous only
            prev_block.size += size
        elif can_merge_with_next:
            # Merge with next only
            current.start = address
            current.size += size
        else:
```

```
            # No merge - insert new free block
            new_block = FreeBlock(address, size)
            self._insert_free_block_after(prev_block, new_block)

    def _remove_free_block(self, block: FreeBlock):
        """Remove a block from the free list"""
        if block.prev:
            block.prev.next = block.next
        else:
            # Removing head
            self.free_list_head = block.next

        if block.next:
            block.next.prev = block.prev

    def _insert_free_block_after(self, prev_block: FreeBlock, new_block: FreeBlock):
        """Insert new_block after prev_block in the free list"""
        if prev_block is None:
            # Insert at head
            new_block.next = self.free_list_head
            if self.free_list_head:
                self.free_list_head.prev = new_block
            self.free_list_head = new_block
        else:
            new_block.next = prev_block.next
            new_block.prev = prev_block
            if prev_block.next:
                prev_block.next.prev = new_block
            prev_block.next = new_block

    def get_free_memory(self) -> int:
        """Helper method to check total free memory"""
        total = 0
        current = self.free_list_head
        while current:
            total += current.size
            current = current.next
        return total

    def get_largest_free_block(self) -> int:
        """Helper method to find largest contiguous free block"""
        max_size = 0
        current = self.free_list_head
        while current:
            max_size = max(max_size, current.size)
            current = current.next
        return max_size
```

Related Problems

- LeetCode 2502: Design Memory Allocator (Simplified version)

- Memory management in operating systems
- Garbage collection algorithms
- Cache replacement policies

Notes

- This problem tests understanding of linked lists, memory management, and algorithm optimization
- Real-world allocators (like `malloc`) use more sophisticated strategies combining multiple techniques
- The interviewer may ask you to start with a simple approach and then optimize based on specific requirements
- Be prepared to trace through your code with examples and prove correctness of your merging logic