**Heap-Based Free Space Management System**

**1. Introduction**

Efficient free space management is a crucial aspect of operating system memory allocation. This document proposes a **heap-based free space management system** that minimizes fragmentation and enables near-constant time allocation. The approach leverages a **max heap** to efficiently track and allocate free memory blocks while ensuring rapid merging of deallocated space.

**2. System Overview**

The system maintains free space using a structure that stores:

* **Total free space available**
* **Pointer to a max heap** storing free memory blocks

Each node in the max heap represents a free memory block and contains:

* **Pointer to the free memory location**
* **Size of the available space**

Additionally, each free memory block contains metadata at both **the start and end**:

* A bit indicating whether it is **allocated or free**
* If free, a **pointer back to the heap node** tracking it

**3. Allocation Mechanism**

**Algorithm for Allocation**

1. Check if the max heap is empty.

2. Extract the largest free block from the max heap.

3. If the block size is greater than or equal to the requested size:

a. If the block size is exactly equal to the requested size:

- Mark the block as allocated in metadata.

b. If the block is larger:

- Split the block into allocated and remaining free space.

- Update metadata of both parts.

- Insert the remaining free part back into the heap.

4. If no suitable block is found, return NULL (memory unavailable).

5. Return the pointer to the allocated memory block.

**Time Complexity:**

* Best case: **O(1)** (direct access to the largest block)
* Worst case: **O(log n)** (heap reorganization after splitting a block)

**4. Deallocation Mechanism**

**Algorithm for Deallocation**

1. Retrieve metadata from the block to check allocation status.

2. Mark the block as free in metadata.

3. Check adjacent memory blocks:

a. If adjacent blocks are free, merge them into a larger block.

b. Update metadata to reflect the new block size.

4. Insert or update the merged block in the max heap.

5. Ensure direct pointer to the heap node is maintained.

**Time Complexity:**

* **O(1)** for direct pointer access to the heap node and merging operations

**5. Advantages**

* **Fast Allocation & Deallocation:** Near **constant-time allocation** and **efficient merging of free blocks**.
* **Minimal Fragmentation:** Metadata tracking at both ends enables **easy coalescing** of adjacent free blocks.
* **Efficient Space Utilization:** The use of a **max heap** ensures that the **largest available block** is used first, reducing fragmentation.
* **Scalability:** Handles dynamic memory requests efficiently, making it suitable for **OS memory management, real-time applications, and database storage systems**.

**6. Challenges and Mitigations**

|  |  |
| --- | --- |
| **Challenge** | **Solution** |
| **Metadata Overhead** | Optimize with compact bit storage and pointer compression |
| **Heap Maintenance** | Use **lazy splitting** and a **Fibonacci heap** for efficiency |
| **Handling Small Requests** | Introduce **segregated free lists** for smaller allocations |
| **Memory Leaks** | Periodically validate metadata integrity to avoid lost free space |
|  |  |

**7. Comparison with Traditional Techniques**

|  |  |  |
| --- | --- | --- |
| Feature | Heap-Based System | Traditional Methods (Bitmap, Linked List) |
| **Allocation Speed** | O(1) best, O(log n) worst | O(n) (bitmap, linked list) |
| **Deallocation Speed** | O(1) | O(n) |
| **Fragmentation** | Low (efficient merging) | Moderate to high |
| **Scalability** | High | Varies by technique |

**8. Comparison with Buddy System**

The **Buddy System** is another commonly used memory allocation technique that divides memory into power-of-two-sized blocks and merges adjacent "buddies" when memory is freed.

**Algorithm for Buddy System Allocation**

1. Find the smallest power-of-two block that can fit the requested size.

2. If an exact match is found, allocate it.

3. If a larger block is found:

a. Split it into two equal "buddy" blocks.

b. Repeat until the block size matches the requested size.

c. Allocate the final block.

**Algorithm for Buddy System Deallocation**

1. Mark the block as free.

2. Check if its buddy block is also free.

3. If the buddy is free, merge them into a larger block.

4. Repeat merging up the hierarchy if possible.

5. Insert the merged block back into the available free list.

|  |  |  |
| --- | --- | --- |
| Feature | Heap-Based System | Buddy System |
| **Allocation Speed** | O(1) best, O(log n) worst | O(log n) |
| **Deallocation Speed** | O(1) | O(log n) |
| **Fragmentation** | Very Low (efficient merging, flexible block sizes) | Higher due to internal fragmentation (power-of-two constraint) |
| **Memory Utilization** | High (no restrictions on block sizes) | Can be inefficient due to forced power-of-two allocations |
| **Merging Efficiency** | Direct pointers allow O(1) merging | O(log n) since buddy lookup is required |
| **Scalability** | High | Moderate (limited by power-of-two constraints) |

**Key Differences:**

* The **heap-based system** provides more **flexible memory allocation** by avoiding power-of-two constraints, leading to **better space utilization**.
* The **buddy system** suffers from **internal fragmentation**, as allocated memory is often larger than required.
* **Merging in the heap-based system is O(1)** due to direct pointers, whereas in the **buddy system, it is O(log n)** since adjacent buddies must be located and validated.
* The heap-based approach is better suited for **highly dynamic memory environments** where fine-grained allocation is required.

**9. Conclusion**

The proposed **heap-based free space management system** offers a highly efficient method for memory allocation and deallocation, minimizing fragmentation while ensuring quick access to free space. By leveraging a **max heap** and **direct pointers for node access**, it significantly improves performance compared to traditional techniques and the **buddy system**.

This approach is well-suited for **operating systems, database management systems, and real-time applications** where efficient memory management is critical.

**Write-Ahead Log (WAL) Design**

**1. Introduction**

A **Write-Ahead Log (WAL)** is a critical component in database management systems, ensuring data integrity and durability in the event of a crash. This document outlines a WAL design where transactions are organized **table-wise**, providing efficient recovery and minimal overhead.

**2. WAL Structure**

**Key Structure**

Each data entry in the WAL is identified using a **composite key**, which consists of:

* **Table Key**: Unique identifier for a table.
* **Row Key**: Unique identifier for a row within the table.

**Storage Mechanism**

* The WAL is **hashed using the table key**, meaning each table has a dedicated WAL table.
* This allows efficient retrieval and recovery for each table independently.
* **Non-Volatile RAM (NVRAM) is used to store WAL**, ensuring persistence without the need for explicit disk flushing.
* For **parallel execution, locking is managed using semaphores** to ensure transaction consistency.

**Entry Format**

Each WAL table entry consists of:

1. **Add/Delete Flag (1 bit)** - Specifies whether the entry is an **addition (1)** or **deletion (0)**.
2. **Data Key (Composite Key)** - Combination of Table Key + Row Key.
3. **Pointer to Actual Data** - A reference to the data stored in the main database or buffer.
4. For commit part there will be pointer to each table that shows current position where further entries will be written. If pointer is behind an entry then that entry is not yet committed if if it is ahead of it then it is committed

**3. WAL Operations**

**Insertion into WAL**

When a transaction modifies data, an entry is added to the WAL before applying changes to the actual database.

**Algorithm:**

1. Compute the hash of the table key to locate the correct WAL table. Use the pointer that is stored with table pointer to find location where next entry will be written

2. Acquire a semaphore lock to ensure exclusive access.

3. Construct a WAL entry:

b. Set Add/Delete Flag based on operation type.

c. Store the composite key (Table Key + Row Key).

d. Store a pointer to the actual data.

4. Append the WAL entry to the respective WAL table.

Increase the pointer

5. Release the semaphore lock.

**Time Complexity:** O(1) (hash lookup and append operation)

**Commit Operation**

After successfully executing a transaction, the corresponding WAL entry is marked as committed.

**Algorithm:**

1. Locate the WAL table using the table key hash.

2. Acquire a semaphore lock for exclusive access.

3. Search for the WAL entry from the last position (newer entries are towards the end).

4. Each WAL table maintains a pointer that does not move until the entry is committed.

6. Since WAL is stored in NVRAM, the commit update is done in-place without additional disk flushing.

7. Release the semaphore lock.

**Time Complexity:** O(1) (hash lookup and update)

**4. Advantages**

* **Efficient Transaction Management**: Table-wise WAL tables improve parallel processing.
* **Fast Lookups**: Hashing ensures O(1) lookup time for WAL entries.
* **Minimal Overhead**: Only a few bits are needed per entry for transaction metadata.
* **Parallel Execution Support**: Semaphore-based locking ensures consistency during concurrent transactions.
* **No Additional Flush Overhead**: Since WAL is stored in **NVRAM**, explicit disk flushing is not required.

**5. Comparison with Traditional WAL Methods**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Proposed WAL Design** | **Traditional WAL (Sequential)** |
| **Lookup Speed** | O(1) (hashing) | O(n) (sequential search) |
| **Commit Handling** | Direct bit update | Sequential log update |
| **Parallel Processing** | High (semaphore-based locking) | Low (single WAL file) |
| **Storage Medium** | NVRAM (no flush needed) | Disk (requires periodic flush) |
| **Scalability** | High (independent WALs per table) | Moderate |

**6. Conclusion**

This WAL design provides an efficient and scalable approach for managing database transactions. By structuring WAL entries **table-wise**, leveraging **hashing**, and utilizing **NVRAM for storage**, the system ensures **fast lookups, minimal overhead, and effective parallel execution**. This makes it well-suited for **large-scale, high-throughput database systems** requiring **fast transaction management** and **quick recovery times**.