**Buddy Memory Allocation System**

*(Documentation with Diagrams, Algorithm, and Code)*

**Introduction**

The **Buddy Memory Allocation** system is a dynamic memory management algorithm that divides memory into partitions, always in powers of two. It efficiently handles allocation and deallocation by splitting and merging memory blocks (buddies), minimizing external fragmentation and allowing fast coalescing of free memory.

**Key Concepts**

* **Power-of-Two Blocks:** Memory is always split into blocks sized as (e.g., 64KB, 128KB, 256KB, ...).
* **Buddy Blocks:** When a block is split, it forms two "buddies"-adjacent blocks of equal size.
* **Binary Tree Representation:** The memory can be visualized as a binary tree, with each split creating two child nodes (buddies).
* **Merging:** When both buddies are free, they merge back into a larger block.

**How It Works**

**Allocation Steps**

1. **Find the smallest block size** () that can fit the requested memory.
2. **Search for a free block** of that size.
3. If not found, **split a larger block** recursively until the required size is available.
4. **Allocate** the block.

**Deallocation Steps**

1. **Mark the block as free.**
2. **Check its buddy:** If the buddy is also free and the same size, **merge** them.
3. **Repeat merging** up the tree as long as possible.

**Diagram: Buddy Allocation Tree**

[1024KB]  
 |  
 / \  
[^512] [^512]  
 | |  
 / \ / \  
[^256][^256][^256][^256]

*Each split divides a block into two buddies. Merging is the reverse.*

**Example**

Suppose you have 1024KB of memory and want to allocate 200KB:

1. The smallest power of two ≥ 200 is 256KB.
2. The allocator splits the 1024KB block into two 512KB buddies, then splits a 512KB into two 256KB buddies.
3. Allocates one 256KB block to the request, leaving the rest free.

**Buddy System Algorithm (Pseudocode)**

**Allocation**

function allocate(request\_size):  
 k = smallest power of 2 ≥ request\_size  
 block = find\_free\_block\_of\_size(k)  
 if block == null:  
 larger\_block = find\_free\_block\_of\_size(>k)  
 while size(larger\_block) > k:  
 split(larger\_block)  
 block = one\_half  
 mark block as allocated  
 return block

**Deallocation**

function deallocate(block):  
 mark block as free  
 while buddy(block) is free and same size:  
 merge(block, buddy(block))  
 block = merged\_block

**C-like Code Example**

// Structure for a memory block  
typedef struct Block {  
 size\_t size;  
 bool free;  
 struct Block\* left;  
 struct Block\* right;  
} Block;  
  
// Allocation function  
Block\* allocate(Block\* root, size\_t request\_size) {  
 if (!root->free || root->size < request\_size)  
 return NULL;  
 if (root->size == request\_size) {  
 root->free = false;  
 return root;  
 }  
 if (!root->left && !root->right) {  
 root->left = create\_block(root->size / 2, true);  
 root->right = create\_block(root->size / 2, true);  
 }  
 Block\* result = allocate(root->left, request\_size);  
 if (!result)  
 result = allocate(root->right, request\_size);  
 root->free = root->left->free && root->right->free;  
 return result;  
}  
  
// Deallocation function  
void deallocate(Block\* block) {  
 block->free = true;  
 // Check and merge with buddy if possible (recursive up the tree)  
}

**JavaScript Example (Simplified)**

class BuddySystem {  
 constructor(size) {  
 this.totalSize = size;  
 this.blocks = [{ start: 0, size: size, free: true, id: 0 }];  
 this.nextId = 1;  
 }  
 allocate(size) {  
 let block = this.findFreeBlock(size);  
 while (block && block.size / 2 >= size) {  
 this.splitBlock(block);  
 block = this.blocks.find(b => b.id === block.id);  
 }  
 if (block) block.free = false;  
 return block;  
 }  
 splitBlock(block) {  
 const newSize = block.size / 2;  
 const newBlock = { start: block.start + newSize, size: newSize, free: true, id: this.nextId++ };  
 block.size = newSize;  
 this.blocks.splice(this.blocks.indexOf(block) + 1, 0, newBlock);  
 }  
 deallocate(blockId) {  
 const block = this.blocks.find(b => b.id === blockId);  
 if (block) block.free = true;  
 this.mergeBlocks(block);  
 }  
 mergeBlocks(block) {  
 // Find buddy and merge if possible  
 }  
 findFreeBlock(size) {  
 return this.blocks.find(b => b.free && b.size >= size);  
 }  
}

**Advantages**

* **Fast allocation and deallocation** (logarithmic time).
* **Efficient merging** of free blocks.
* **Low external fragmentation**.

**Drawbacks**

* **Internal fragmentation:** Always rounds up to the next power of two.
* **Memory overhead:** Needs extra data structures to track splits and merges.

**References**

* [Wikipedia: Buddy memory allocation][[1]](#fn1)
* [GingerBill: Memory Allocation Strategies][[2]](#fn2)
* [Tidjma: Buddy Memory Allocation][[3]](#fn3)
* [GitHub: Buddy-System-Memory-Allocation-1][[4]](#fn4)

**Summary Table**

|  |  |  |
| --- | --- | --- |
| Step | Action | Result |
| Allocation | Split blocks as needed | Allocated block (power of 2) |
| Deallocation | Mark block free, merge | Larger free blocks |

**Buddy System Diagram**

Initial: [1024KB Free]  
Split: [^512][^512]  
Split: [^256][^256][^512]  
Allocate: [A:256][^256][^512] (A is allocated)  
Deallocate: [^256][^256][^512] (A freed)  
Merge: [^512][^512]  
Merge: [^1024]

**The buddy system is a classic, efficient memory allocation technique that uses splitting and merging of power-of-two blocks, often visualized as a binary tree. It is widely used in operating systems and memory managers for its speed and simplicity.**

⁂

1. <https://en.wikipedia.org/wiki/Buddy_memory_allocation>

1. <https://www.gingerbill.org/article/2021/12/02/memory-allocation-strategies-006/>

1. <https://www.tidjma.tn/en/electrical/buddy-memory-allocation-/>

1. <https://github.com/Vatsalsoni13/Buddy-System-Memory-Allocation-1>