Title: Implementing Buddy Algorithm

Introduction:

Storage management routines of a programming environment are crucial tools in programs dealing with complex data structures such as lists or trees. Inside most operating systems,similar routines are used for such applications as allocating primary memory to hold segments in segmented virtual memory systems, allocating space for the code and variables of user programs, and allocating the system data structures themselves.

The storage region from which memory is dynamically allocated in response to arbitrary user demand is usually referred to as the heap. The term heap clearly indicates the relative disorder of the allocation and deallocation requests, as opposed, for example, to the orderly pattern of allocation and deallocation imposed by the use of a stack. The routines for allocation and deallocation from the heap are frequently referred to as the heap manager component of the system or language which they support.

This project implements a buddy system, which is an allocator that allocates memory within a fixed linear address range. It spans the address range with a binary tree that tracks free space.When memory is allocated,nodes in the tree are split recursively until a node of the appropriate size is reached. Every split results in two child nodes, each of which is the buddy of the other. When a node is freed, the node and its buddy can be merged again if the buddy is also free.

Problem Description:

The problem statement as explained above basically deals with efficiently using the memory so as to reduce the internal & external fragmentation but, what is the issue that the old method had ?

The older methods used the concept of Fixed size allocation which means the extra space allocated will get allocated even if the requirement is less, which causes a lot of internal fragmentation. In the buddy system the heap manager maintains enough information to determine which blocks in the heap are free at any time. Typically, this

information will be stored in the form of a linked list or binary tree, where the first word of each free block holds the address of the next free block.

Solution Approach:

To allocate requested memory we will find the next greater power of 2 and find it in the allocation pool(in implementation binary tree is used) where free memory nodes are available in size of power of 2.
We will define a level for the binary tree which will be capable of allocation 2^level number of memory blocks.Example:If we take level=7,we can have 2^7=128 total memory blocks to allocate.
Some of the terminologies used:

- Index:Index represents the block number.
- Offset:Offset is weight that a node contains which inturn helps in finding an address.
- Level:Level at which the node is placed in the tree.

Each node of allocation pool or binary tree is denoted as one of the below:

- UNUSED - 0
- USED - 1
- FULL - 3
- SPLIT - 2

1.UNUSED- If a node is not allocated it will be marked as UNUSED.Initially all the nodes will be marked as UNUSED.

2.USED-If a node is allocated it will be marked as USED.

3.FULL-If both the childs of a node are USED then the node will be marked as FULL.

4.SPLIT:If any one of the child of a node is available to allocate memory,then the node will be marked as SPLIT.

Work Distribution:

Member 1:Vishal Pandey
-memalign
-posix_memalign
-Address Computation

Member 2:Mayank Mukundam
-malloc
-Buddy index computation
-combine

Member 3:Yash Sharma
-mallinfo
-realloc
- Internal Fragmentation calculation

Member 4:Bhupendra Sharma
-free
-calloc

Problems Faced and Shortcomings:

- **Collaboratively work on single code**

One of the problems that we faced is how to work on single code collaboratively and incorporate changes and being in sync at same time.

- **Implementational Problem**

Since there were very few resources by which we can actually get an idea of implementation of this memory allocation technique,we faced some issues while implementing the theoretical idea.

- **Implementation of Coalescing**

Every node in the allocation tree has to follow some property,so to incorporate that property after allocation/deallocation we have to maintain a lot of attributes which was a challenging part.

- **Research took time**

We have to put in a lot of time to do research and to get an idea about the project, we were left with a limited amount of time to do the implementation part.

Learnings:

- **We can allocate/deallocate memory very fast:**

We actually have a memory allocation technique that can allocate memory as fast as log(NUM_OF_MEMORY_BLOCKS).Let say if we have 1024 memory blocks,we can build a binary tree of 10 level and to allocate any memory block we can find and allocate free block
In logn time.So we got to learn about an efficient way of memory allocation.

- **How to work as a Team:**

We have individually done a lot of projects but we have learned how to collaborate and work as a team.

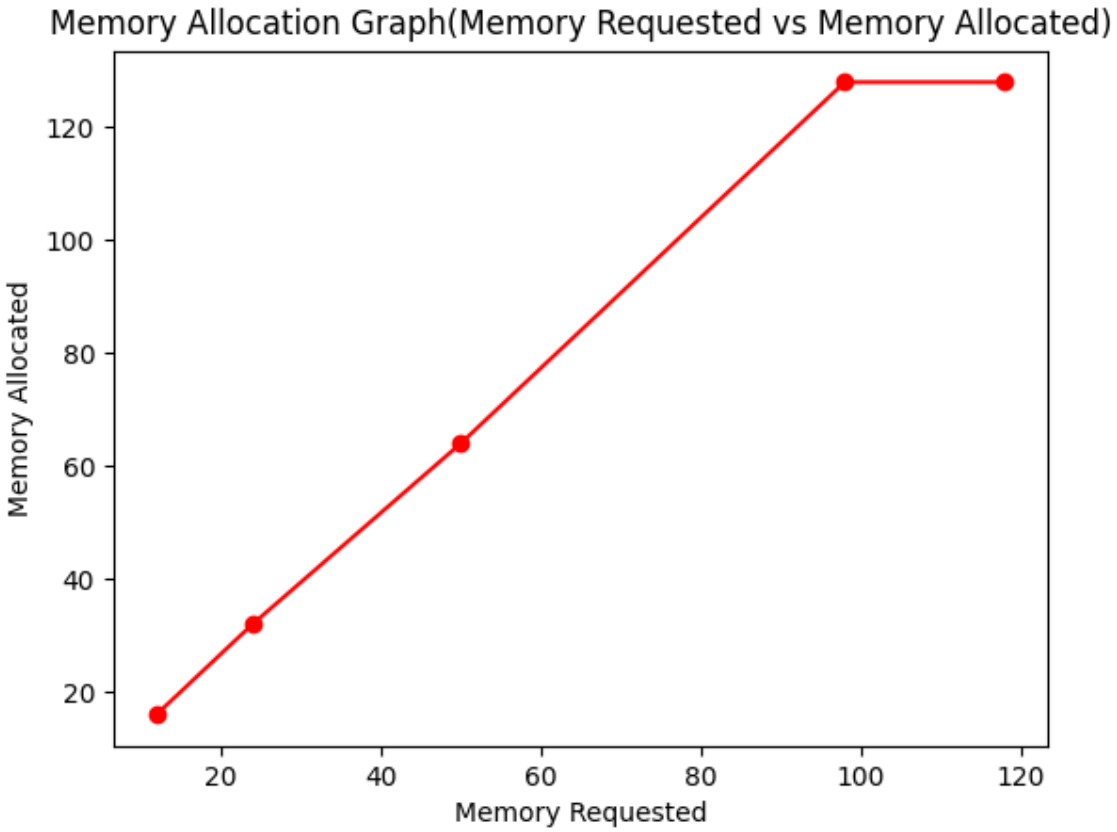
- **How to work on single code and be in-sync**

We learned how to collaborate and be in sync about different modules and components that are being built individually.

Test Cases:

- We have written extra assertion tests to check our program running efficiently.
- It helped us to maintain a sync between what is expected & what is the result.

Result:



Conclusion:

We have learnt about the fast working of memory allocation algorithm with this system along with reduction in Internal Fragmentation.The cost to allocate/deallocate memory is low compared to other memory allocation techniques due to use of coalescing.