Heap Memory Manager

Course: CSL3030

Members:

- ➢ Barun Shakya (B19CSE020)
- ➢ Harpal Sahil Santosh (B19CSE107)
- ➢ Ankur Borkar (B19CSE025)
- ➢ Chirag (B19CSE026)

Introduction

In Programming world, dynamic memory allocation and deallocation are of utmost use. The structures defined by users need to be dynamically allocated and deallocated upon user requests. In C-Programs Linux uses std C-library functions malloc for memory allocation and free for memory deallocation. The memory is allocated in the Heap Segment.

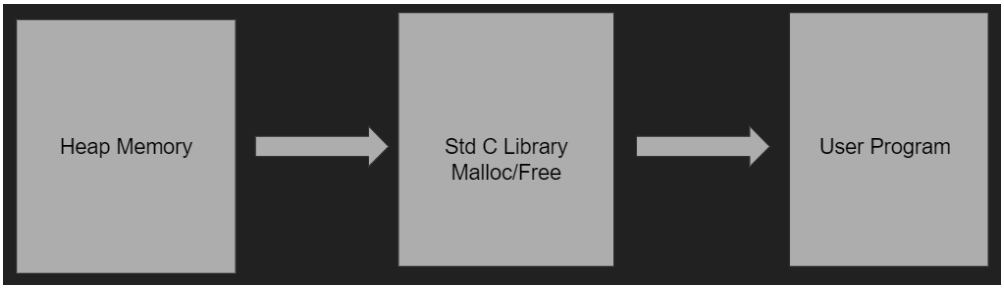
In this project, we have replaced the std C-library with our newly implemented Virtual Memory Manager that runs on top of Linux virtual manager. Memory allocation and deallocation happens with the use of xmalloc and xfree functions implemented in our Virtual Memory Manager. Our Virtual Memory Manager keeps track of all the structures to whom memory is allocated. It also keeps track of the free blocks available in all the vm pages for particular structures. The free blocks are managed using priority queue, the block having largest size is kept at the front of the doubly linked priority queue. The memory allocation takes place at the front block, i.e, the block of largest size. Our virtual memory manager also deals with the issue of memory fragmentation.

Objectives

- Handle memory allocation and deallocation request by applications
- Catch memory leaks
- Display the amount of memory allocated to every structure which are currently in use
- Handle memory fragmentation problem
- Design and implement malloc and free functions

Methodology

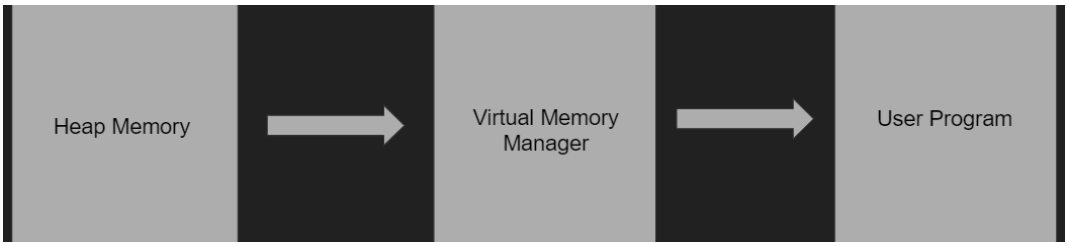
Virtual Memory Manager-Standard C-Library



The basic flow of memory to the User program is as follows:

- Firstly, the heap memory manager allocates virtual memory pages to the standard c-library.
- Then, the standard c-library allocates/deallocates memory to the user program on requests.

Now, we have implemented our own memory manager with greater functionalities that run on top of the Linux virtual memory manager.



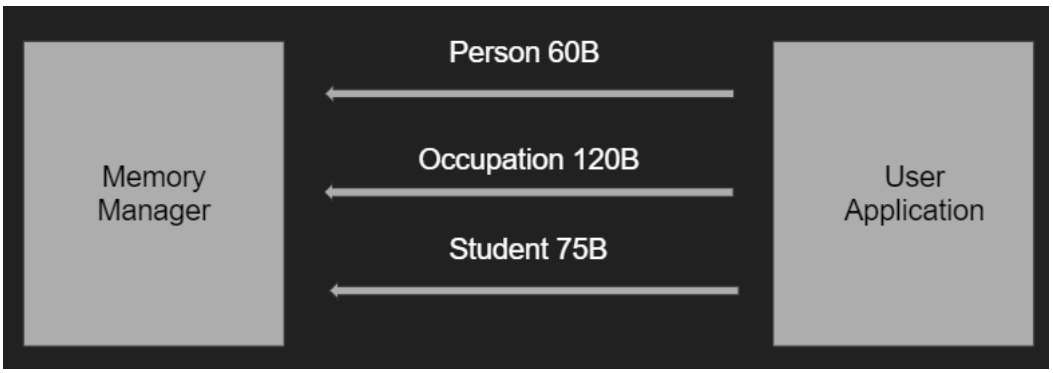
The Virtual memory manager is a replacement to the existing c-library functionalities.
Here we will request the full VM Page from the heap memory and allocate the small chunk to the user programs. Our virtual memory manager also keeps track of all the structures to whom the memory is allocated. It also keeps track of free memory blocks available in the virtual pages. It deals with internal memory fragmentation by merging the continuous blocks.

- Whenever a program requests memory dynamically using malloc function call, then it requests to heap memory for a full VM page.
- Then, this program allocates a small chunk of the VM page to the user program whenever it requests some memory.
- Whenever a free function call is executed, the memory allocated is released, The memory released is returned to the heap.

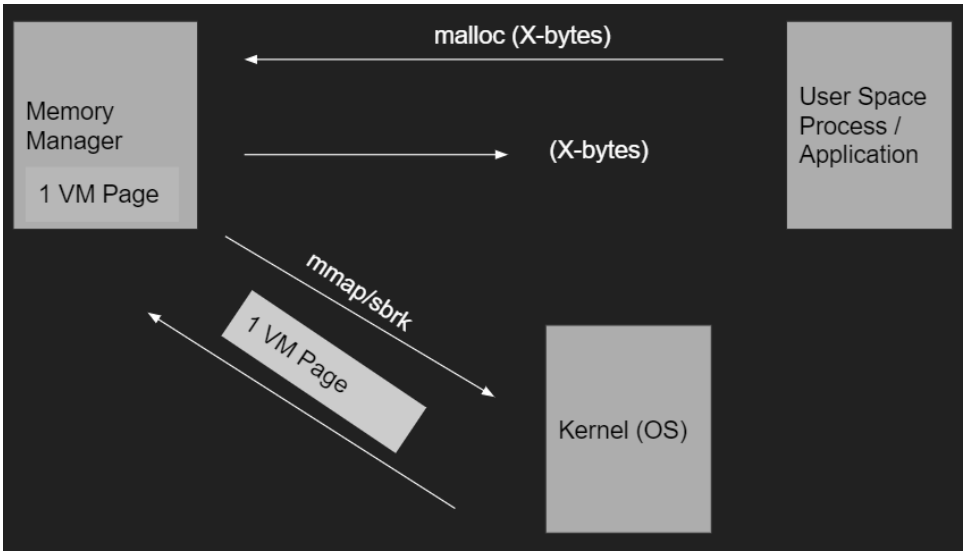
Page Family (Structures) Registration

Whenever an application requests memory for a particular structure, then that structure needs to be registered with the virtual memory manager.
The structure used for registration consists of the structure's name, size and pointer to the first VM page. It is called page family. VM page stores the page, starting from bottom to top in a contiguous manner. The VM

page used to store page families is called VM page for families. Multiple VM pages for families are linked through a linked list, the most recent one at the head.
The information about structures is stored in the virtual page itself.



Project Architecture



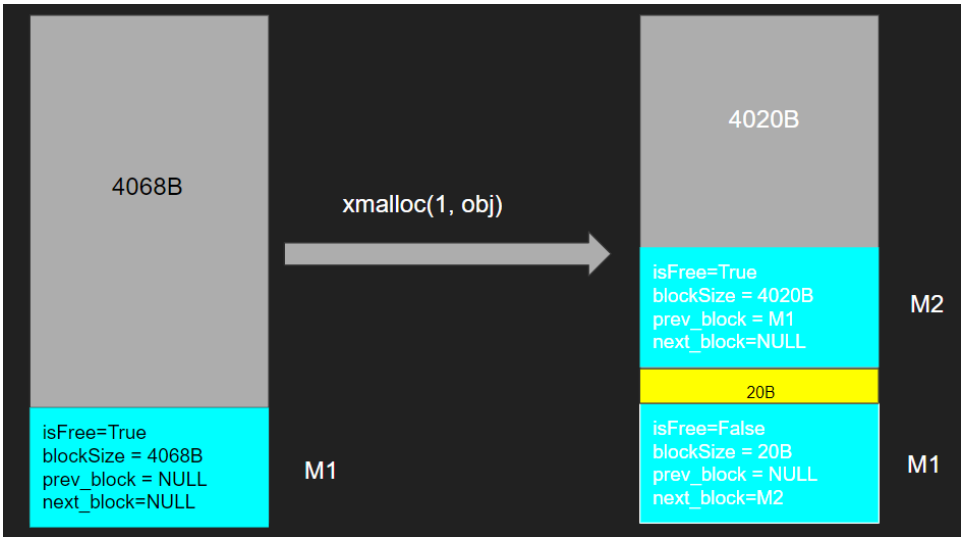
Users can request allocation of any X-bytes of memory. Memory allocation between Memory Manager and Kernel happens in the units of Page Sizes. Kernel Allocates 1 VM Page to the Memory Manager and then memory manager allocates X-amount of memory to the User Program. The kernel function used for VM page allocation is sbrk and mmap.
If the process requests for more memory allocation and if that amount of memory is available in the earlier allocated VM page, then the memory manager allocates the memory to the user process. When the process frees all the memory of the given VM page, then the page is returned back to the kernel. The kernel functions we have used for vm page deallocation to the kernel are brk and munmap.

Meta and Data Blocks

Data block contains the data for which the application has requested access for.
Data Block is a part of the vm page given to applications to store the data.
Data Block has Meta Block associated.
Meta Block contains the meta information of the data blocks.
Meta Block Structure has following Features:

- Data Block Size: The size of the data block that is allocated to the user program.
- Previous Block: The pointer to the previous meta block. The previous pointer of the first meta block is NULL.
- Next Block: The pointer to the next data block.
- Is Free: The block is free or occupied, If the block is allocated memory for the user program then is free is set to FALSE.
- Offset: Offset is the memory distance of the meta block from the bottom of the vm page.

Splitting Data Blocks:

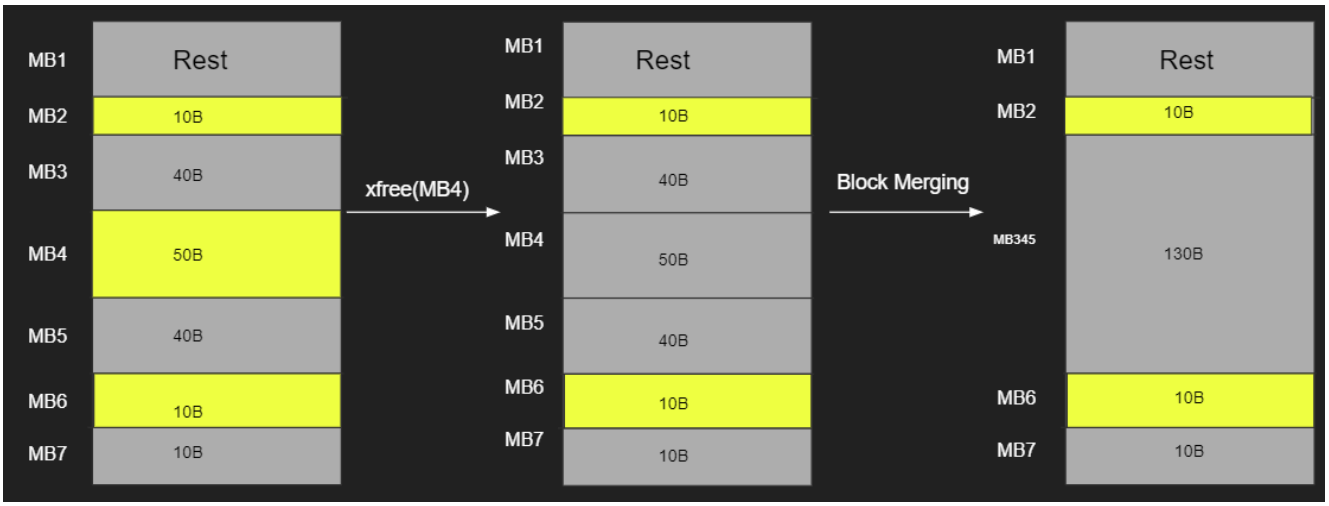


When our linux memory manager requests a fresh virtual memory page from the kernel for memory allocation to the application then that complete memory virtual page is actually available to satisfy the memory request of the application. It means that this entire memory page will act as a big data block. So we know every data block.is guarded by its metablock therefore as soon as our linux memory management requests fresh data virtual memory space from the kernel. Our linux memory manager will treat this new virtual memory space as a free data block. So what we need is to create our metablock marked as true and this metabock is the guardian of a data block whose size is 4068 bytes and this metablock is also lowest meta block present in this virtual memory page and this metablock also highest meta block present in this virtual memory page.

prev_block = NULL and next_block = NULL (initially)

- 4068 B is size of metablock
- 4068 B is available in order to satisfy a new malloc request from the application. As soon as our linux memory manager requests from the kernel, we will treat that particular virtual memory page as one big data block.

Data Block Merging

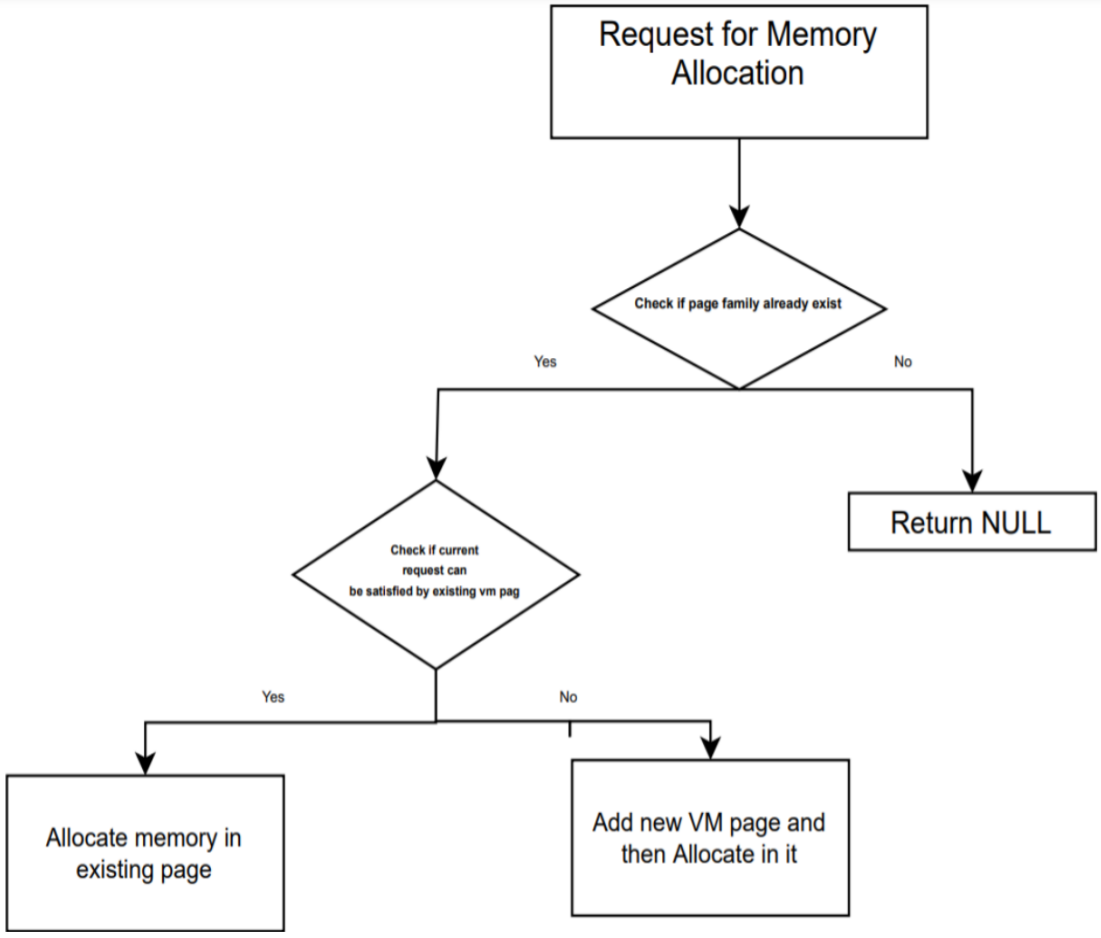


- When the application release the block of memory, linux memory manager memory need to perform Block Merging
- Block merging done using Block Merging Algorithm which states that if more than one data blocks are empty in a continuous location in the virtual memory manager then they merge together to form a big data form.

Priority Queue

- All the free data blocks are ordered using priority queue.
- The free data block having the largest size is at the front of the queue.
- The memory requested is satisfied by the largest block, block at the front.
- The block is splitted and the queue is adjusted to bring the largest block at the front and insert the remaining block from the splitted block at the adequate place.

Flow Chart For Memory Allocation



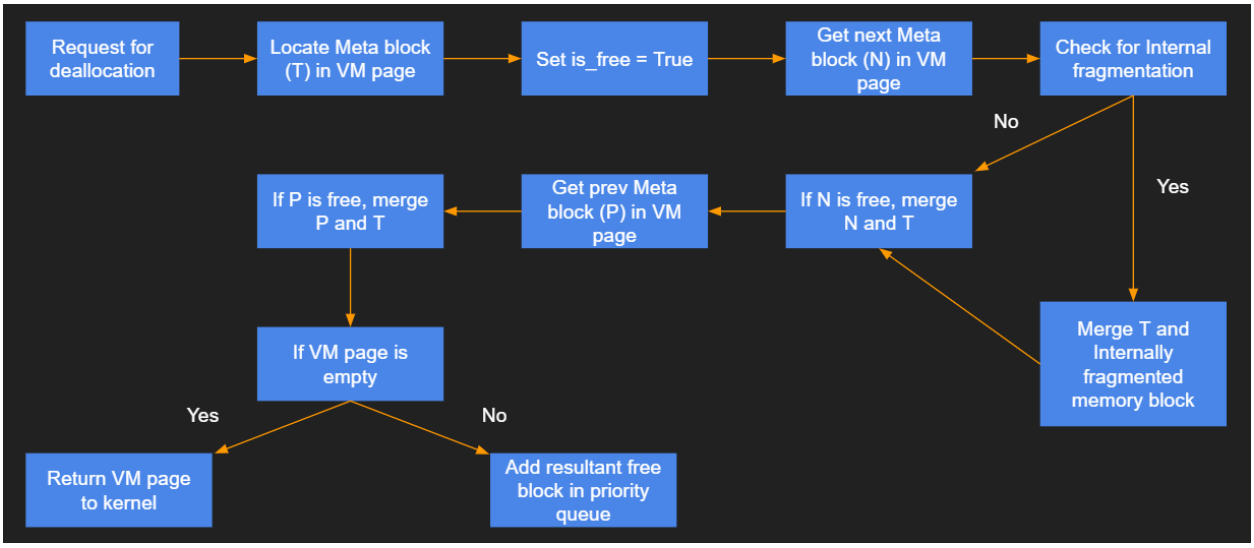
Steps

- Linux memory manager request for the memory allocation
- It will check if the page family already exist or not
    - If doesn't exit
        - Return NULL
    - If exist
        - Then check if current request can be satisfied by existing virtual memory page
            - If does not satisfied
                - Add new VM page and allocate in it
            - If satisfied
                - Allocate memory in existing page

Deallocation of memory:

For this we have implemented our own free() function i.e. xfree(). **void xfree(void * ptr)** is a signature of function.It takes one argument that is void * ptr. Here ptr is a pointer to the data block that needs to be free.

Flowchart

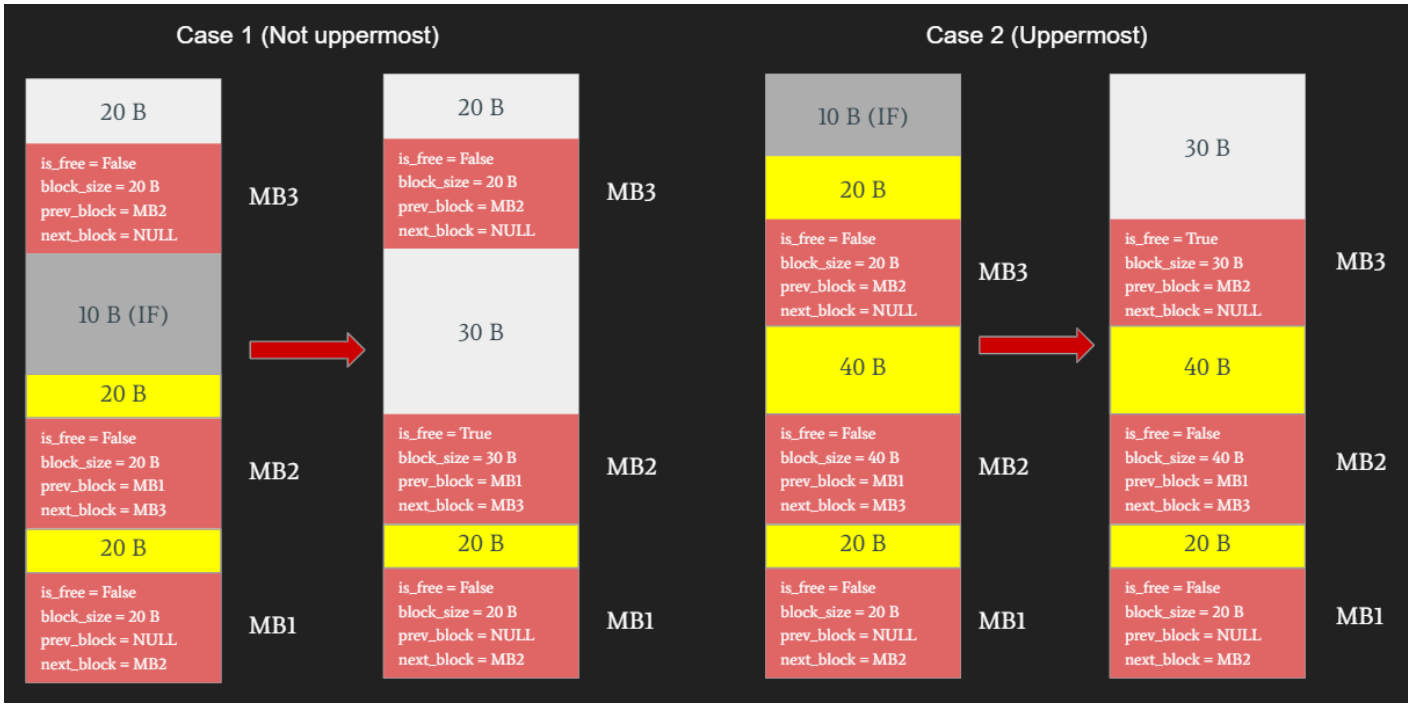


First application invokes the xfree routine. Then the Memory manager will locate that particular meta block (T) and set is_free status = true. After this it gets the next meta block (N) and checks whether internally fragmented memory is present or not. If It is present then merges that internally fragmented memory and targeted data block which result in formation of larger free data block. After handling this internally fragmented memory, the memory manager will check the is_free status of the next meta block (N), if it is free to merge N and T. Similarly it will check the is_free status of the previous meta block (P) and merges P and T. After performing this merging operation, the manager will check whether the resultant VM page is empty or not. If it is empty then manager will return that page to kernel else add that resultant free block to our priority queue which keeps record of all free data blocks

Here handling the Internally fragmented memory is quite tricky. There are 2 possible scenarios of Internal fragmentation. First scenario would be, there exists a block of free memory between 2 occupied blocks. And the free memory is sufficient to accommodate one meta block but corresponding data block's size is not sufficient to allocate to structure and another scenario would be, the size of free memory is not sufficient to accommodate a single meta block and hence memory remains unallocatable.

First scenario can be easily handled with the algorithm we have designed. But we are required to handle the second scenario explicitly.

In the second scenario itself we have 2 cases. First if the Data block to be freed is not the uppermost block and second is the data block to be freed is the uppermost data block in the VM page.



- Yellow block - Occupied data block
- Red block - Meta block
- White block - Free data block
- Grey block - Internally fragmented block

Simple check we added for

- ❖ case 1 (This case will occur when targeted metablock next_block != NULL) is ,

offset(targeted meta block)+size(meta block)+size(targeted data block) < offset(next meta block)

And for

- ❖ case 2 (This case will occur when for targeted metablock next_block = NULL) is,

offset(targeted meta block)+size(meta block)+size(targeted data block) < size(VM page)-head address(VM page)

If true then we can say there exists IF memory. We are handling this by adjusting next and previous pointers of meta blocks.

Results

For testing our Heap Memory Manager, we have created a test file which has 2 structs. The 1st struct name is ‘t’ and the 2nd struct name is ’s’. In the first case, we have created 3 instances of ‘t’: t1,t2 & t3, and 2 instances of ‘s’: s1 & s2.
Memory statistics after the execution of 1st case is shown below:

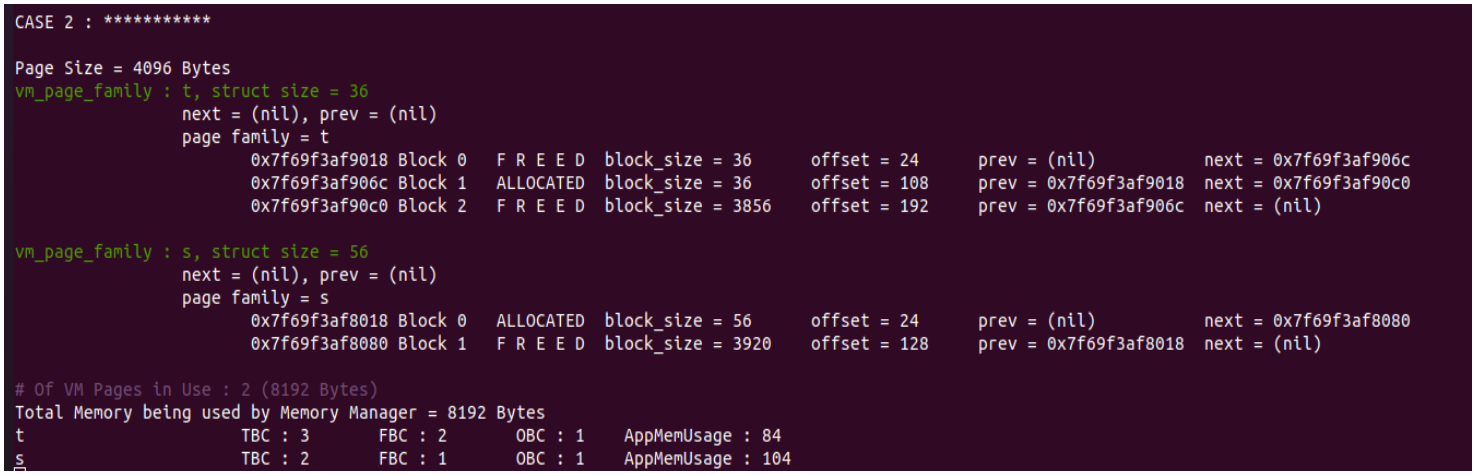
```
chirag@Chirag:~/Desktop/os/project$ ./test_app.exe
Page Family : t, Size = 36
Page Family : s, Size = 56

CASE 1 : ***********

Page Size = 4096 Bytes
vm_page_family : t, struct size = 36
                next = (nil), prev = (nil)
                page family = t
                       0x7f69f3af9018 Block 0   ALLOCATED  block_size = 36      offset = 24      prev = (nil)           next = 0x7f69f3af906c
                       0x7f69f3af906c Block 1   ALLOCATED  block_size = 36      offset = 108     prev = 0x7f69f3af9018  next = 0x7f69f3af90c0
                       0x7f69f3af90c0 Block 2   ALLOCATED  block_size = 36      offset = 192     prev = 0x7f69f3af906c  next = 0x7f69f3af9114
                       0x7f69f3af9114 Block 3   F R E E D  block_size = 3772    offset = 276     prev = 0x7f69f3af90c0  next = (nil)

vm_page_family : s, struct size = 56
                next = (nil), prev = (nil)
                page family = s
                       0x7f69f3af8018 Block 0   ALLOCATED  block_size = 56      offset = 24      prev = (nil)           next = 0x7f69f3af8080
                       0x7f69f3af8080 Block 1   ALLOCATED  block_size = 56      offset = 128     prev = 0x7f69f3af8018  next = 0x7f69f3af80e8
                       0x7f69f3af80e8 Block 2   F R E E D  block_size = 3816    offset = 232     prev = 0x7f69f3af8080  next = (nil)

# Of VM Pages in Use : 2 (8192 Bytes)
Total Memory being used by Memory Manager = 8192 Bytes
t                    TBC : 4       FBC : 1       OBC : 3    AppMemUsage : 252
s                    TBC : 3       FBC : 1       OBC : 2    AppMemUsage : 208
```

Observations:

1. Struct ‘t’ has size 36 bytes and struct ‘s’ has size 56 bytes.
2. 1 virtual page size is 4096 bytes.
3. As the virtual pages in our HMM are connected through a doubly linked list, next = (nil) and prev = (nil) indicates that only 1 virtual page is used for struct ‘t’.
4. The 1st column represents the starting address of the meta block associated with the data block. The 2nd column shows the block number.
5. The 3rd column shows whether the data block is allocated or is free. The 4th column shows the data block size. Every data block has its meta block of fixed size of 48 bytes.
6. The 5th column shows the offset of the meta block from the starting address of the virtual page. The first 24 bytes of the virtual page are used for storing necessary information about the virtual page.
7. The 6th column shows the address of the previous meta block. The 7th column shows the address of the next meta block.

The results show that Block 0,1 and 2 of the 1st page are allocated for 3 instances of ‘t’ struct, and Block 0 and 1 of the 2nd page are allocated for the 2 instances of ‘s’ struct.
The total space usage for 3 instances of ‘t’ struct is 36*3(data block) + 48*3(meta block) = 252 bytes.
The total space usage for 2 instances of ‘s’ struct is 56*2(data block) + 48*2(meta block) = 208 bytes.

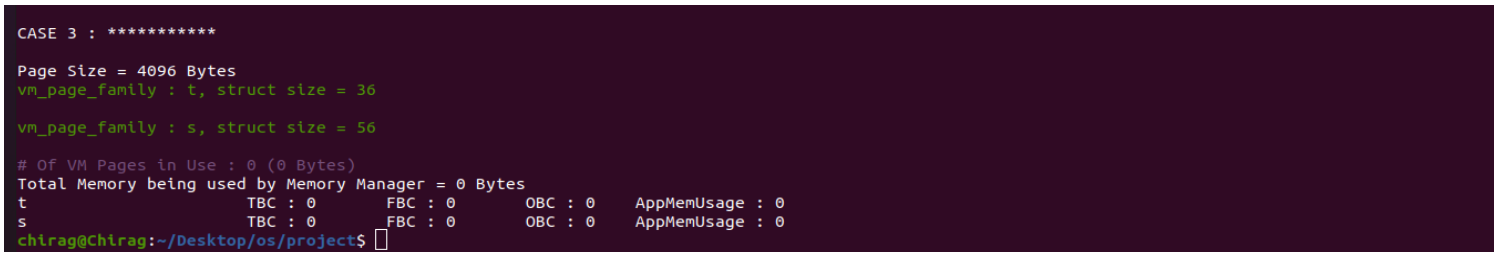
In the 2nd case, we have freed t1,t3 and s2. Memory statistics after the execution of the 2nd case is shown below:



The results show that for the 1st first page, Block 0 is freed and Block 2 is freed and merged with Block 3 to form a big Block 2. For the 2nd page, Block 1 is freed and merged with Block 2 to form a big Block 1. Now,

The total space usage for 1 instances of ‘t’ struct is 36*1(data block) + 48*1(meta block) = 84 bytes.
The total space usage for 2 instances of ‘s’ struct is 56*1(data block) + 48*1(meta block) = 104 bytes.

In the 3rd case, we have free all the memory. Memory statistics after the execution of the 3rd case is shown below:



The results show that all the memory is being freed and the virtual pages are being returned to the kernel by our heap memory manager as the total memory used by memory manager shows 0 bytes in the image.

Conclusion

We have successfully designed and implemented our own virtual heap memory manager which can perform allocation and deallocation of heap memory. Apart from this our heap manager also shows statistics of memory allocation for every structure. We have identified and handled memory fragmentation problems explicitly. Outputs shown in the result section shows the correctness of our memory manager.