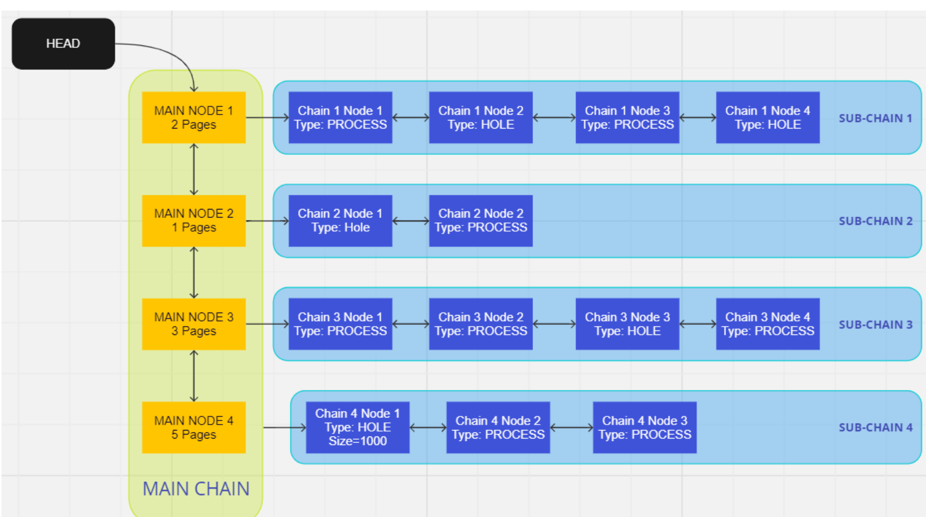
OS-Assignment-3(write-up)

Github link (private) -
https://github.com/sahilguptasg2017/OS-assignments-/tree/main/operating_system/Assignment-3

We have used a 2-D linked list for the implementation of Memory Management System (Mems) . We have made two linked list chains . One is the main chain and the other corresponds to the subchain for each main node in the main chain .
We have not used any other library management system functions other than mmap and munmap .
Size allocation for mmap and munmap is done in multiples of PAGE_SIZE only. and we are ensuring that memory is not wasted anywhere weather it is when we are creating main chain nodes or sub chain nodes using mmap. We are also making one to one mapping from memes virtual address to mems physical address .We are handling the edge cases (including the one that two holes merge into one hole if they are consecutively present) .

Free List Structure

We are making a 2-D linked list in which there are two linked list present , that is for each main chain node in the first linked list there exists a subchain linked list for it . Its structure looks similar to this .



We are saving the virtual start and virtual end address in the struct of both the main chain node and the struct of subchain node .
We then use this to get the mems virtual address from mems_malloc when the user asks for the size allocation .

The headers files included are :

```c
#include <stdint.h>
#include<stdio.h>
#include<stdlib.h>
#include <sys/mman.h>
```

The structs used for the main chain and sub chain are :

```c
typedef struct main_node{

   struct main_node* prev ;
   int pages ;
   unsigned long virtual_start ;
   unsigned long virtual_end ;
   void* phy_addr ;
   struct subchain_node* subchain ;
   struct main_node* next ;

}main_node;

typedef struct subchain_node{
   struct subchain_node* prev ;
   int type ; // 0 for hole 1 for process ..
   unsigned long virtual_start ;
   unsigned long virtual_end ;
   void* phys_addr ;
   size_t size ;
   struct subchain_node* next ;

}subchain_node;
```

They are also typedef for easy usage .

The global variables created are as follows :

```
unsigned long vir_address ;
struct main_node* head = NULL;
struct main_node* current_pointer = NULL;
unsigned long  main_node_size = sizeof(struct main_node) ;
unsigned long subchain_size = sizeof(struct subchain_node) ;
struct subchain_node* current_pointer_subchain = NULL;
struct main_node* init_main = NULL;
struct subchain_node* init_sub = NULL;
int chain_count = 1;
int subchain_count_ls = 1 ;
```

Now explaining the coding parts and the 6 functions made for the Mems :

1) void mems_init(): In this function we are initializing our virtual address which is starting from 1000 , our head which we create using mmap(that is the first mmap allocation and this is fully used to create the other main chain nodes till this space gets exhausted , current pointer : which marks where in the memory currently the last node present is that is if this current pointer goes out of range a new mmap is done so that a new main node can be formed , the initial pointer which marks the starting of the main chain memory allocation address .
   The mems_init() function goes like :

```
void mems_init(){
   vir_address = 1000 ;
   head = (main_node*)mmap(NULL, PAGE_SIZE*1,
PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE , -1, 0);
   if(head == MAP_FAILED){
       perror("mmap failed");
   }

head->next = NULL ;
   current_pointer=head ;
   init_main = head ;
  // printf("%d\n" , rounded_val(PAGE_SIZE)) ;
   //printf("%d\n",rounded_val(PAGE_SIZE)/main_node_size) ;
   //printf("%d\n",rounded_val(PAGE_SIZE)/main_node_size  - 1 ) ;
//    printf("%lu",(unsigned long)current_pointer) ;

  // printf("%lu\n",sizeof(subchain_node));
```

```
//  printf("%lu\n",sizeof(main_node));
}
```

2) <u>void mems_finish():</u> This function is called at the end of the MeMs system and unallocate the allocated memory using munmap present in the main chain node that is the memory the user asked in the mems_malloc .
So, we just munmap all the stored physical memory in the main node struct and at the end make the head as NULL so that no main chain is present now . The size of the physical memory will be the (number of pages present in the main chain node) * PAGE_SIZE . So, the goes for mems_finish() goes like :

```
void mems_finish(){
   // main_node* start = head ;
   // while (start != NULL){
   //     main_node* temp = start ;
   //     munmap(start,start->pages * PAGE_SIZE) ;
   //     subchain_node* subchain_start = temp->subchain ;
   //     while(subchain_start != NULL){
   //         subchain_node* temp1 = subchain_start ;
   //         munmap(subchain_start , ) ;
   //         subchain_start = temp1->next ;
   //     }
   //     start = temp->next ;
   // }
   main_node* start  = head->next ;
   while(start != NULL){
       if(munmap(start->phy_addr , start->pages * PAGE_SIZE) == -1){
           perror("munmap failed");
       }
       start = start->next ;
   }
   head = NULL ;

}
```

3) <u>void* mems_malloc(size_t size):</u> In this the user asks for the size to be allocated and the Mems returns the Mems virtual address . Firstly it iterates through the structure and if a HOLE is found of sufficiently large size it returns the Mems virtual address(in void *) .
So first we initialize the flag as 0 and number of pages required as 1 .
Now if the size required is 0 , the function simply returns NULL as no space is allocated in the Mems system.
Now if the size is not equal to 0 , if the head is not NULL that is there is a structure formed , we find if there is sufficient large HOLE and if it satisfies , it will form 2 cases : that are if the size of the node is greater than the size needed by the user and if the size is just equal to the size of the node .
Now if the size of the node is greater than the size of the size asked by the user , the HOLE is divided into a smaller HOLE and a PROCESS . The code for the same goes like :

```
       if(head->next != NULL){
           //printf("head != NULL\n") ;
           main_node* main_chain = head ;
           //printf("%lu\n" , main_chain->phy_addr) ;
           while (main_chain!=NULL) {
               subchain_node* sub_chain = main_chain->subchain;
               //need to increase the current_pointer //
               while (sub_chain!=NULL) {
                   if(sub_chain->type == 0 && sub_chain->size >
size){
                       subchain_node* new_sub = NULL ;
                       new_sub = create_new_subchain(new_sub) ;
                       unsigned long virt_end =
sub_chain->virtual_end ;
                       size_t sub_size = sub_chain->size;
                       sub_chain->virtual_end =
sub_chain->virtual_start + size - 1 ;
                       sub_chain->size = size ;
                       sub_chain->type = 1 ;
                       sub_chain->next = new_sub ;
                       new_sub->prev = sub_chain ;
                       new_sub->type = 0 ;
                          new_sub->phys_addr = sub_chain->phys_addr
+ sub_chain->size ;
                       new_sub->size = sub_size - size ;
```

```
                       new_sub->virtual_start =
sub_chain->virtual_start + size ;
                       new_sub->virtual_end = virt_end ;
                       new_sub->next = NULL;
                       flag = 1 ;
                       return (void*) sub_chain->virtual_start ;

                   }
                   else if (sub_chain->type == 0 &&
sub_chain->size==size){
                       //printf("dqwadwq") ;
                       sub_chain->type = 1 ;
                       flag = 1 ;
                       return (void*)sub_chain->virtual_start;
                   }
                   sub_chain= sub_chain->next ;
               }
               main_chain = main_chain->next ;
           }
```

Now if there is no such node with sufficient space , we create a new main node using the function `main_node* create_new_main_node(main_node* node)`
So it creates a new node and it allocate the page for the size , the user asked .
Now we create the new subchain for the main chain node using `subchain_node* create_new_subchain(subchain_node* new_sub)`
 Function and then give it the mems virtual address and mems physical address .
The code for the two helper function goes like :

```

main_node* create_new_main_node(main_node* node){

  // printf("this is first term: %u  \n",(unsigned
long)current_pointer+main_node_size) ;
  // printf("this is second term: %u  \n",(unsigned
long)init_main+PAGE_SIZE) ;
   //main_node* node = NULL ;
   if((unsigned long)current_pointer+main_node_size < (unsigned
long)init_main+(unsigned long)PAGE_SIZE &&
chain_count%(rounded_val(PAGE_SIZE)/main_node_size -1)!=0 ){
       node = (main_node*)((unsigned char*)current_pointer +
main_node_size);
```

```
        current_pointer = node;
        chain_count ++ ;
       // printf("%d\n",chain_count) ;
        //printf("gg\n") ;
        return node ;
    }
    else{
      //printf("here\n") ;
        node = (main_node*)mmap(NULL, PAGE_SIZE*1,
PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE ,-1,0) ;
        if(node == MAP_FAILED){
            perror("mmap failed");
        }
        chain_count++ ;
    //    printf("%d" , chain_count) ;
         init_main = node;
        current_pointer = node;
        node = (main_node*)((unsigned char*)current_pointer +
main_node_size);
        current_pointer = node ;
        return node ;
    }
}
```

```
subchain_node* create_new_subchain(subchain_node* new_sub){
    //subchain_node* new_sub = NULL ;
    if((unsigned long)current_pointer_subchain + subchain_size <=
(unsigned long)init_sub + PAGE_SIZE &&
subchain_count_ls%(rounded_val(PAGE_SIZE)/subchain_size  - 1 ) !=
0){
        new_sub =(subchain_node* )((unsigned
char*)current_pointer_subchain+subchain_size) ;
        current_pointer_subchain = new_sub ;
        subchain_count_ls ++ ;
        //printf("%d\n",subchain_count_ls) ;
        return new_sub ;
    }
    else {
        //printf("here\n") ;
```

```
       new_sub =(subchain_node*
)mmap(NULL,PAGE_SIZE*1,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVAT
E ,-1,0) ;
       if(new_sub == MAP_FAILED){
           perror("mmap failed");
       }
       subchain_count_ls++ ;
      // printf("%d\n",subchain_count_ls);
    init_sub = new_sub ;
       current_pointer_subchain = new_sub ;
       new_sub =(subchain_node* )((unsigned
char*)current_pointer_subchain+subchain_size) ;
       current_pointer_subchain = new_sub ;
       return new_sub ;
   }
}
```

While creating a new main chain node or a new subchain node , we also round of the value of PAGE_SIZE to nearest ceil of multiple of 4096(as mmap works on the multiple of 4096) .

```
int rounded_val(int x){
   if(x%4096 == 0){
       return x ;
   }
   else{
       return ((x/4096) + 1) *4096 ;
   }



}
```

So the code for this case goes like :

```
if(flag == 0){
               main_node* new_main = NULL ;
               new_main = create_new_main_node(new_main);
               main_node* main_chain = head ;
               while(main_chain->next != NULL){
                   main_chain = main_chain->next ;
```

```
            }

            void* temp =
mmap(NULL,n*PAGE_SIZE,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE
, -1, 0);
            new_main->pages = n ;
            new_main->phy_addr = temp;
            if(new_main->phy_addr == MAP_FAILED){
                perror("mmap failed") ;
            }
           //printf("%zu\n",new_main->phy_addr);
             // reach the last main node and then add a new main
node and do the things  ..
            main_chain->next = new_main ;
            new_main->prev = main_chain ;
            new_main->virtual_start = new_main->prev->virtual_end
+ 1 ;

            new_main->virtual_end = new_main->virtual_start +
n*PAGE_SIZE - 1 ;
            subchain_node* new_subchain = NULL ;
            new_subchain = create_new_subchain(new_subchain) ;
            new_main->subchain = new_subchain ;
            new_main->next = NULL;
            new_subchain->size = n*PAGE_SIZE ;
            new_subchain->phys_addr = new_main->phy_addr  ;
            new_subchain->type = 0;
            new_subchain->size = n*PAGE_SIZE ;
            new_subchain->virtual_start = new_main->virtual_start
;

            new_subchain->virtual_end = new_main->virtual_end ;

            if(new_subchain->size > size){
                subchain_node* newest_subchain = NULL;
                newest_subchain =
create_new_subchain(newest_subchain) ;
                new_subchain->next = newest_subchain ;
                newest_subchain->prev = new_subchain ;
                new_subchain->virtual_end =
new_subchain->virtual_start + size - 1 ;
                new_subchain->size = size ;
```

```
                    new_subchain->type = 1 ;
                    newest_subchain->phys_addr =
newest_subchain->prev->phys_addr + size  ;
                    newest_subchain->virtual_start =
newest_subchain->prev->virtual_start + size ;
                    newest_subchain->size = new_main->pages*PAGE_SIZE
- size ;
                    newest_subchain->type = 0 ;
                    newest_subchain->next = NULL;
                    return (void*) new_subchain->virtual_start ;
                }
                else if( new_subchain->size == size){
                    new_subchain->type = 1 ;
                    return (void*)new_subchain->virtual_start ;
                }
            }
        }
```

Now for the first case if there is no node present we handle the case in which head.next == NULL .
We make the first node of the main chain and and assign the first node as `(main_node* )((unsigned char*)current_pointer+main_node_size)` that is we just increase the pointer and assign the first node there .
Then we do the mmap to store the physical address of the node that is with

```
          first_node->phy_addr = mmap(NULL,n*PAGE_SIZE,
PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE ,-1,0) ;
```

This gives the physical memory(mems_physiacal address) that the user will get .
Now we set the VA of the main node and its corresponding virtual start and virtual end .
We now make its first subchain using the first mmap as

```
          subchain_node* subchain =
(subchain_node*)mmap(NULL,PAGE_SIZE*1,PROT_READ|PROT_WRITE,MAP_ANONY
MOUS|MAP_PRIVATE ,-1,0) ;
```

Now we set its corresponding values and then check the two conditions similarly as above .
The whole code goes like :

```
            else{
            //printf("dfkoaed\n");
           // printf("%lu %lu\n",current_pointer, main_node_size) ;
           //printf("%c" , current_pointer) ;
            main_node* first_node = NULL ;

            first_node = (main_node* )((unsigned
char*)current_pointer+main_node_size);

            // else{
            //     first_node = mmap(NULL,PAGE_SIZE
,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE ,-1,0) ;
            //     init_main = first_node ;
            //     current_pointer = first_node ;  // will do this for the
adding of main node also when flag = 0 ;
            // }
           // printf("%lu\n",first_node) ;
            current_pointer = first_node ;
            first_node->phy_addr = mmap(NULL,n*PAGE_SIZE,
PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE ,-1,0) ;
            if(first_node->phy_addr == MAP_FAILED){
                perror("mmap failed") ;
            }
            //printf("%zu\n",first_node->phy_addr) ;
            //printf("fefe");
            head->next = first_node ;
            first_node->prev = head ;
            first_node->pages = n ;
            first_node->virtual_start = vir_address;
            first_node->virtual_end = first_node->
virtual_start+n*PAGE_SIZE - 1 ;
            vir_address+=n*PAGE_SIZE ;
            subchain_node* subchain =
(subchain_node*)mmap(NULL,PAGE_SIZE*1,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|M
AP_PRIVATE ,-1,0) ;
            if(subchain == MAP_FAILED){
                perror("mmap failed") ;
            }
            first_node->next = NULL;
            first_node->subchain = subchain ;
```

```
            subchain->phys_addr = first_node->phy_addr  ;
            subchain->size = n*PAGE_SIZE ;
            subchain->type = 0 ;
            subchain->virtual_start = first_node->virtual_start ;
            subchain->virtual_end = first_node->virtual_end ;
            subchain->next = NULL;
          //  printf("%lu\n",subchain->virtual_start);
          //  printf("%lu\n",subchain->virtual_end) ;
            current_pointer_subchain = subchain ;
            init_sub = subchain ;
            if(subchain->size > size){
                subchain_node* new_sub = NULL ;
                new_sub = create_new_subchain(new_sub) ;
                unsigned long virt_end = subchain->virtual_end ;
                subchain->virtual_end = subchain->virtual_start+ size - 1 ;
                subchain->size = size ;
                subchain->type = 1;
                subchain->next = new_sub ;
                new_sub->prev = subchain ;
                new_sub->size = (first_node->pages * PAGE_SIZE) - size ;
                new_sub->virtual_start = subchain->virtual_start + size ;
                new_sub->virtual_end = virt_end ;
                new_sub->type = 0;
                new_sub->next = NULL;
                new_sub->phys_addr = subchain->phys_addr + subchain->size ;
                return (void*)subchain->virtual_start ;
            }
            else if(subchain->size == size){
                subchain->type = 1 ;
                return (void*)subchain->virtual_start ;
            }
        }
    }
}
```

So the whole mems_malloc code goes like :

```
void* mems_malloc(size_t size){

   main_node* curr_main = head->next ;
```

```
    while(curr_main != NULL){
        subchain_node*  curr_sub = curr_main->subchain ;
        while(curr_sub!=NULL){
            curr_sub->virtual_end = curr_sub->virtual_start +
curr_sub->size - 1;
            curr_sub = curr_sub->next ;
        }
        curr_main = curr_main->next ;
    }



    int flag = 0 ;
    int n = 1 ;
    while(n*PAGE_SIZE < size){
        n++ ;
    }

    //printf("mems_mall\n");

    if(size == 0){
        printf("please give a size\n") ;
        return NULL;
    }
    else{
        //printf("etre\n");
        if(head->next != NULL){
            //printf("head != NULL\n") ;
            main_node* main_chain = head ;
            //printf("%lu\n" , main_chain->phy_addr) ;
            while (main_chain!=NULL) {
                subchain_node* sub_chain = main_chain->subchain;
                //need to increase the current_pointer //
                while (sub_chain!=NULL) {
                    if(sub_chain->type == 0 && sub_chain->size > size){
                        subchain_node* new_sub = NULL ;
                        new_sub = create_new_subchain(new_sub) ;
                        unsigned long virt_end = sub_chain->virtual_end ;
                        size_t sub_size = sub_chain->size;
```

```
                        sub_chain->virtual_end = sub_chain->virtual_start +
size - 1 ;
                        sub_chain->size = size ;
                        sub_chain->type = 1 ;
                        sub_chain->next = new_sub ;
                        new_sub->prev = sub_chain ;
                        new_sub->type = 0 ;
                           new_sub->phys_addr = sub_chain->phys_addr +
sub_chain->size ;
                        new_sub->size = sub_size - size ;
                        new_sub->virtual_start = sub_chain->virtual_start +
size ;

                        new_sub->virtual_end = virt_end ;
                        new_sub->next = NULL;
                        flag = 1 ;
                        return (void*) sub_chain->virtual_start ;

                    }
                    else if (sub_chain->type == 0 &&
sub_chain->size==size){
                        //printf("dqwadwq") ;
                        sub_chain->type = 1 ;
                        flag = 1 ;
                        return (void*)sub_chain->virtual_start;
                    }
                    sub_chain= sub_chain->next ;
                }
                main_chain = main_chain->next ;
            }
            if(flag == 0){
                main_node* new_main = NULL ;
                new_main = create_new_main_node(new_main);
                main_node* main_chain = head ;
                while(main_chain->next != NULL){
                    main_chain = main_chain->next ;
                }

                void* temp =
mmap(NULL,n*PAGE_SIZE,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE , -1,
0);
```

```
                new_main->pages = n ;
                new_main->phy_addr = temp;
                if(new_main->phy_addr == MAP_FAILED){
                    perror("mmap failed") ;
                }
               //printf("%zu\n",new_main->phy_addr);
                 // reach the last main node and then add a new main node
and do the things  ..
                main_chain->next = new_main ;
                new_main->prev = main_chain ;
                new_main->virtual_start = new_main->prev->virtual_end + 1 ;
                new_main->virtual_end = new_main->virtual_start +
n*PAGE_SIZE - 1 ;
                subchain_node* new_subchain = NULL ;
                new_subchain = create_new_subchain(new_subchain) ;
                new_main->subchain = new_subchain ;
                new_main->next = NULL;
                new_subchain->size = n*PAGE_SIZE ;
                new_subchain->phys_addr = new_main->phy_addr  ;
                new_subchain->type = 0;
                new_subchain->size = n*PAGE_SIZE ;
                new_subchain->virtual_start = new_main->virtual_start ;
                new_subchain->virtual_end = new_main->virtual_end ;

                if(new_subchain->size > size){
                    subchain_node* newest_subchain = NULL;
                    newest_subchain = create_new_subchain(newest_subchain)
;
                    new_subchain->next = newest_subchain ;
                    newest_subchain->prev = new_subchain ;
                    new_subchain->virtual_end = new_subchain->virtual_start
+ size - 1 ;
                    new_subchain->size = size ;
                    new_subchain->type = 1 ;
                    newest_subchain->phys_addr =
newest_subchain->prev->phys_addr + size  ;
                    newest_subchain->virtual_start =
newest_subchain->prev->virtual_start + size ;
                    newest_subchain->size = new_main->pages*PAGE_SIZE -
size ;
```

```
                    newest_subchain->type = 0 ;
                    newest_subchain->next = NULL;
                    return (void*) new_subchain->virtual_start ;
                }
                else if( new_subchain->size == size){
                    new_subchain->type = 1 ;
                    return (void*)new_subchain->virtual_start ;
                }
            }
        }
        else{
            //printf("dfkoaed\n");
           // printf("%lu %lu\n",current_pointer, main_node_size) ;
           //printf("%c" , current_pointer) ;
            main_node* first_node = NULL ;

            first_node = (main_node* )((unsigned
char*)current_pointer+main_node_size);

            // else{
            //     first_node = mmap(NULL,PAGE_SIZE
,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE ,-1,0) ;
            //     init_main = first_node ;
            //     current_pointer = first_node ;  // will do this for the
adding of main node also when flag = 0 ;
            // }
           // printf("%lu\n",first_node) ;
            current_pointer = first_node ;
            first_node->phy_addr = mmap(NULL,n*PAGE_SIZE,
PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE ,-1,0) ;
            if(first_node->phy_addr == MAP_FAILED){
                perror("mmap failed") ;
            }
            //printf("%zu\n",first_node->phy_addr) ;
            //printf("fefe");
            head->next = first_node ;
            first_node->prev = head ;
            first_node->pages = n ;
            first_node->virtual_start = vir_address;
```

```
            first_node->virtual_end = first_node->
virtual_start+n*PAGE_SIZE - 1 ;
            vir_address+=n*PAGE_SIZE ;
            subchain_node* subchain =
(subchain_node*)mmap(NULL,PAGE_SIZE*1,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|M
AP_PRIVATE ,-1,0) ;
            if(subchain == MAP_FAILED){
                perror("mmap failed") ;
            }
            first_node->next = NULL;
            first_node->subchain = subchain ;
            subchain->phys_addr = first_node->phy_addr  ;
            subchain->size = n*PAGE_SIZE ;
            subchain->type = 0 ;
            subchain->virtual_start = first_node->virtual_start ;
            subchain->virtual_end = first_node->virtual_end ;
            subchain->next = NULL;
        //  printf("%lu\n",subchain->virtual_start);
        //  printf("%lu\n",subchain->virtual_end) ;
            current_pointer_subchain = subchain ;
            init_sub = subchain ;
            if(subchain->size > size){
                subchain_node* new_sub = NULL ;
                new_sub = create_new_subchain(new_sub) ;
                unsigned long virt_end = subchain->virtual_end ;
                subchain->virtual_end = subchain->virtual_start+ size - 1 ;
                subchain->size = size ;
                subchain->type = 1;
                subchain->next = new_sub ;
                new_sub->prev = subchain ;
                new_sub->size = (first_node->pages * PAGE_SIZE) - size ;
                new_sub->virtual_start = subchain->virtual_start + size ;
                new_sub->virtual_end = virt_end ;
                new_sub->type = 0;
                new_sub->next = NULL;
                new_sub->phys_addr = subchain->phys_addr + subchain->size ;
                return (void*)subchain->virtual_start ;
            }
            else if(subchain->size == size){
                subchain->type = 1 ;
```

```
                return (void*)subchain->virtual_start ;
            }
        }
    }
}
```

4) <u>void mems_free(void* ptr):</u>

In this we pass the mems_virtual address in the function and the functions frees that subchain node . That is if the type of the subchain is PROCESS , it is changed to HOLE and if it is HOLE it remains as a HOLE .

Now if in this process if a case arrises such that there are two HOLES consecutively , the two HOLES are merged using the `merge_holes` function .

The merge_holes function goes like :

```
void merge_holes(subchain_node* subchain){

    if(subchain->next != NULL){
        while(subchain->next->type == 0){
            subchain->size = subchain->size + subchain->next->size ;

            subchain->virtual_end = subchain->next->virtual_end ;

            subchain->next = subchain->next->next ;
            if(subchain->next == NULL){
                break;
            }
        }
    }

}
```

To find the subchain with the same virtual address as given by the void* ptr we use the function

```
void finding_subchain
```

The code for the same goes like :

```
void finding_subchain(void*v_ptr){
    main_node* main_start = head->next ;
    while(main_start!=NULL){
        subchain_node* subchain_start = main_start->subchain ;
        while(subchain_start!=NULL){
```

```
            // printf("%u %u \n",(unsigned long)v_ptr
,subchain_start->virtual_start ) ;
            if((unsigned long)v_ptr ==
subchain_start->virtual_start){
                subchain_start->type = 0 ;
                return ;
            }
            subchain_start = subchain_start->next ;
        }
        main_start = main_start->next ;
    }

}
```

Now using these two functions in our mems_free function . The code goes like :

```
void mems_free(void *v_ptr){

    finding_subchain(v_ptr);


    main_node* curr_main = head->next ;
    while(curr_main!=NULL){
        subchain_node* curr_subchain = curr_main->subchain ;
        while(curr_subchain != NULL){
            if(curr_subchain->type == 0){
                merge_holes(curr_subchain) ;

            }
            curr_subchain = curr_subchain->next ;
        }
        curr_main = curr_main->next ;
    }
}
```

5) <u>void mems_print_stats():</u>
This function just prints the Mems status such that if head is NULL it just prints

```
        printf("Pages used:    %d\n",0) ;
        printf("Space unused:  %d\n",0) ;
        printf("Main Chain Length:    %d\n",0) ;
```

```
        printf("Sub-Chain Length array: []\n") ;
```

And returns
Else we just traverse the structure to get the desired output .
The whole code goes like :

```
void mems_print_stats(){
   printf("---------MeMS SYSTEM STATS------------\n");
   if(head == NULL){
        printf("Pages used:    %d\n",0) ;
        printf("Space unused:  %d\n",0) ;
        printf("Main Chain Length:    %d\n",0) ;
        printf("Sub-Chain Length array: []\n") ;
        return ;
   }
   main_node* curr_main = head->next ;
   while(curr_main != NULL){
        subchain_node*  curr_sub = curr_main->subchain ;
        while(curr_sub!=NULL){
            curr_sub->virtual_end = curr_sub->virtual_start +
curr_sub->size - 1;
            curr_sub = curr_sub->next ;
        }
        curr_main = curr_main->next ;
   }
   //int counter = 0  ;
   //int arr[999999] ;
   //int num = 0 ;
   int pages = 0 ;
   int mem_unused = 0;
   main_node* curent_main_node = head->next ;
   while(curent_main_node != NULL){
        subchain_node* current_subchain_node =
curent_main_node->subchain ;
        while(current_subchain_node!=NULL){
            if(current_subchain_node->type == 0){
                mem_unused += current_subchain_node->size ;
            }
         //   num++ ;
            current_subchain_node = current_subchain_node->next ;
        }
        pages += curent_main_node->pages ;
```

```
        //arr[counter++] = num ;
        //num = 0;
        curent_main_node = curent_main_node->next ;
    }


    int main_chain_len = 0 ;
    main_node* current_main = head->next;
    while(current_main!=NULL){

printf("MAIN[%lu:%lu]->",current_main->virtual_start,current_main->v
irtual_end) ;
        subchain_node* current_subchain = current_main->subchain ;
        while(current_subchain!=NULL){
            if(current_subchain->type == 0){

printf("H[%lu:%lu]<->",current_subchain->virtual_start ,
current_subchain->virtual_end);
            }
            else{

printf("P[%lu:%lu]<->",current_subchain->virtual_start ,
current_subchain->virtual_end);
            }
            current_subchain = current_subchain->next ;
        }
        printf("NULL\n") ;
        main_chain_len+=1 ;
        current_main = current_main->next ;
    }
    printf("Pages used:    %d\n",pages) ;
    printf("Space unused:  %d\n",mem_unused) ;
    printf("Main Chain Length:    %d\n",main_chain_len) ;
    printf("Sub-Chain Length array: [") ;
    main_node* curr_mainnode = head->next ;
    while(curr_mainnode !=NULL){
        subchain_node* curr_subchain = curr_mainnode->subchain ;
        int subchain_size = 0 ;
        while(curr_subchain != NULL) {
            subchain_size++;
```

```
            curr_subchain = curr_subchain->next ;
        }
        printf("%d, ",subchain_size) ;
        curr_mainnode = curr_mainnode->next ;
    }
    // printf("Sub-Chain Length array: [") ;
    // for(int i = 0 ; i < counter ; i++){
    //     printf("%d, ",arr[i]);
    // }
    printf("]\n");
    printf("----------------------------------------\n");
}
```

6) <u>void *mems_get(void*v_ptr):</u>
This gives the mems_physical address for the mems_virtual address passed to this function . This function iterates through the structure and if the v_ptr lies in the range of virtual start and virtual end of the subchain , the function returns the physical address of the subchain, if nothing is found it simply returns NULL .
The code for the same goes like :

```
void *mems_get(void*v_ptr){

    main_node* current_main = head->next ;
    // while(current_main != NULL){
    //     printf("%zu\n" , current_main->phy_addr) ;

    //     current_main = current_main->next ;
    // }
    void* phys_addr_for_vptr = NULL ;
    //return head->next->phy_addr ;
    while(current_main!=NULL){
        if(current_main->virtual_start<=(unsigned long)v_ptr &&
current_main->virtual_end>=(unsigned long)v_ptr){
            phys_addr_for_vptr = (void*)((unsigned
long)current_main->phy_addr + (unsigned long) v_ptr -
current_main->virtual_start) ;
            break;
        }
        current_main = current_main->next ;
```

```
    }

    return phys_addr_for_vptr;


}
```

Test Case 1 : (Given on Github repo) :
Input :

```
// include other header files as needed
#include"mems.h"


int main(int argc, char const *argv[])
{
    // initialise the MeMS system
    mems_init();
    int* ptr[10];

    /*
    This allocates 10 arrays of 250 integers each
    */
    printf("\n------- Allocated virtual addresses [mems_malloc]
-------\n");
    for(int i=0;i<10;i++){
        ptr[i] = (int*)mems_malloc(sizeof(int)*250);
        printf("Virtual address: %lu\n", (unsigned long)ptr[i]);
    }

    /*
    In this section we are tring to write value to 1st index of
array[0] (here it is 0 based indexing).
    We get get value of both the 0th index and 1st index of array[0]
by using function mems_get.
    Then we write value to 1st index using 1st index pointer and try
to access it via 0th index pointer.

    This section is show that even if we have allocated an array
using mems_malloc but we can
```

```
    retrive MeMS physical address of any of the element from that
array using mems_get.
    */
    printf("\n------ Assigning value to Virtual address [mems_get]
-----\n");
    // how to write to the virtual address of the MeMS (this is given
to show that the system works on arrays as well)
    int* phy_ptr= (int*) mems_get(&ptr[0][1]); // get the address of
index 1
    phy_ptr[0]=200; // put value at index 1
    int* phy_ptr2= (int*) mems_get(&ptr[0][0]); // get the address of
index 0
    printf("Virtual address: %lu\tPhysical Address: %lu\n",(unsigned
long)ptr[0],(unsigned long)phy_ptr2);
    printf("Value written: %d\n", phy_ptr2[1]); // print the address
of index 1

    /*
    This shows the stats of the MeMS system.
    */
    printf("\n--------- Printing Stats [mems_print_stats]
--------\n");
    mems_print_stats();

    /*
    This section shows the effect of freeing up space on free list
and also the effect of
    reallocating the space that will be fullfilled by the free list.
    */
    printf("\n--------- Freeing up the memory [mems_free]
--------\n");
    mems_free(ptr[3]);
    mems_print_stats();
    ptr[3] = (int*)mems_malloc(sizeof(int)*250);
    mems_print_stats();

    printf("\n--------- Unmapping all memory [mems_finish]
--------\n\n");
    mems_finish();
    return 0;
```

```
}
/
```

```
Output :
------- Allocated virtual addresses [mems_malloc] -------
Virtual address: 1000
Virtual address: 2000
Virtual address: 3000
Virtual address: 4000
Virtual address: 5096
Virtual address: 6096
Virtual address: 7096
Virtual address: 8096
Virtual address: 9192
Virtual address: 10192

------ Assigning value to Virtual address [mems_get] -----
Virtual address: 1000   Physical Address: 140188468871168
Value written: 200

--------- Printing Stats [mems_print_stats] --------
---------MeMS SYSTEM STATS------------
MAIN[1000:5095]->P[1000:1999]<->P[2000:2999]<->P[3000:3999]<->P[4000
:4999]<->H[5000:5095]<->NULL
MAIN[5096:9191]->P[5096:6095]<->P[6096:7095]<->P[7096:8095]<->P[8096
:9095]<->H[9096:9191]<->NULL
MAIN[9192:13287]->P[9192:10191]<->P[10192:11191]<->H[11192:13287]<->
NULL
Pages used: 3
Space unused:  2288
Main Chain Length:     3
Sub-Chain Length array: [5, 5, 3, ]
----------------------------------------

--------- Freeing up the memory [mems_free] --------
---------MeMS SYSTEM STATS------------
MAIN[1000:5095]->P[1000:1999]<->P[2000:2999]<->P[3000:3999]<->H[4000
:5095]<->NULL
MAIN[5096:9191]->P[5096:6095]<->P[6096:7095]<->P[7096:8095]<->P[8096
:9095]<->H[9096:9191]<->NULL
MAIN[9192:13287]->P[9192:10191]<->P[10192:11191]<->H[11192:13287]<->
NULL
Pages used: 3
Space unused:  3288
Main Chain Length:     3
Sub-Chain Length array: [4, 5, 3, ]
----------------------------------------
---------MeMS SYSTEM STATS------------
```

```
MAIN[1000:5095]->P[1000:1999]<->P[2000:2999]<->P[3000:3999]<->P[4000
:4999]<->H[5000:5095]<->NULL
MAIN[5096:9191]->P[5096:6095]<->P[6096:7095]<->P[7096:8095]<->P[8096
:9095]<->H[9096:9191]<->NULL
MAIN[9192:13287]->P[9192:10191]<->P[10192:11191]<->H[11192:13287]<->
NULL
Pages used: 3
Space unused:  2288
Main Chain Length:     3
Sub-Chain Length array: [5, 5, 3, ]
----------------------------------------

--------- Unmapping all memory [mems_finish] --------
```

The two are merging in the first chain after mems_free .

Test 2 : -
Input :

```
// include other header files as needed
#include"mems.h"


int main(int argc, char const *argv[])
{
   // initialise the MeMS system
   mems_init();
   int* ptr[74];

   /*
   This allocates 10 arrays of 250 integers each
   */
   printf("\n------- Allocated virtual addresses [mems_malloc]
-------\n");
   for(int i=0;i<74;i++){
       ptr[i] = (int*)mems_malloc(sizeof(int)*4095);
       printf("Virtual address: %lu\n", (unsigned long)ptr[i]);
   }

   /*
   In this section we are tring to write value to 1st index of array[0]
(here it is 0 based indexing).
```

```
   We get get value of both the 0th index and 1st index of array[0] by
using function mems_get.
   Then we write value to 1st index using 1st index pointer and try to
access it via 0th index pointer.

   This section is show that even if we have allocated an array using
mems_malloc but we can
   retrive MeMS physical address of any of the element from that array
using mems_get.
   */
   printf("\n------ Assigning value to Virtual address [mems_get]
-----\n");
   // how to write to the virtual address of the MeMS (this is given to
show that the system works on arrays as well)
   int* phy_ptr= (int*) mems_get(&ptr[0][1]); // get the address of index
1
   phy_ptr[0]=200; // put value at index 1
   int* phy_ptr2= (int*) mems_get(&ptr[0][0]); // get the address of index
0
   printf("Virtual address: %lu\tPhysical Address: %lu\n",(unsigned
long)ptr[0],(unsigned long)phy_ptr2);
   printf("Value written: %d\n", phy_ptr2[1]); // print the address of
index 1

   /*
   This shows the stats of the MeMS system.
   */
   printf("\n--------- Printing Stats [mems_print_stats] --------\n");
   mems_print_stats();

   /*
   This section shows the effect of freeing up space on free list and also
the effect of
   reallocating the space that will be fullfilled by the free list.
   */
   printf("\n--------- Freeing up the memory [mems_free] --------\n");
   mems_free(ptr[3]);
   mems_print_stats();
   ptr[3] = (int*)mems_malloc(sizeof(int)*250);
   mems_print_stats();
```

```c
    printf("\n--------- Unmapping all memory [mems_finish] --------\n\n");
    mems_finish();
    return 0;
}
```

Output :

```
------- Allocated virtual addresses [mems_malloc] -------
Virtual address: 1000
Virtual address: 17384
Virtual address: 33768
Virtual address: 50152
Virtual address: 66536
Virtual address: 82920
Virtual address: 99304
Virtual address: 115688
Virtual address: 132072
Virtual address: 148456
Virtual address: 164840
Virtual address: 181224
Virtual address: 197608
Virtual address: 213992
Virtual address: 230376
Virtual address: 246760
Virtual address: 263144
Virtual address: 279528
Virtual address: 295912
Virtual address: 312296
Virtual address: 328680
Virtual address: 345064
Virtual address: 361448
Virtual address: 377832
Virtual address: 394216
Virtual address: 410600
Virtual address: 426984
Virtual address: 443368
Virtual address: 459752
Virtual address: 476136
Virtual address: 492520
Virtual address: 508904
Virtual address: 525288
Virtual address: 541672
Virtual address: 558056
Virtual address: 574440
Virtual address: 590824
```

```
Virtual address: 607208
Virtual address: 623592
Virtual address: 639976
Virtual address: 656360
Virtual address: 672744
Virtual address: 689128
Virtual address: 705512
Virtual address: 721896
Virtual address: 738280
Virtual address: 754664
Virtual address: 771048
Virtual address: 787432
Virtual address: 803816
Virtual address: 820200
Virtual address: 836584
Virtual address: 852968
Virtual address: 869352
Virtual address: 885736
Virtual address: 902120
Virtual address: 918504
Virtual address: 934888
Virtual address: 951272
Virtual address: 967656
Virtual address: 984040
Virtual address: 1000424
Virtual address: 1016808
Virtual address: 1033192
Virtual address: 1049576
Virtual address: 1065960
Virtual address: 1082344
Virtual address: 1098728
Virtual address: 1115112
Virtual address: 1131496
Virtual address: 1147880
Virtual address: 1164264
Virtual address: 1180648
Virtual address: 1197032

------ Assigning value to Virtual address [mems_get] -----
Virtual address: 1000   Physical Address: 139761110110208
Value written: 200

--------- Printing Stats [mems_print_stats] --------
---------MeMS SYSTEM STATS------------
MAIN[1000:17383]->P[1000:17379]<->H[17380:17383]<->NULL
MAIN[17384:33767]->P[17384:33763]<->H[33764:33767]<->NULL
MAIN[33768:50151]->P[33768:50147]<->H[50148:50151]<->NULL
```

```
MAIN[50152:66535]->P[50152:66531]<->H[66532:66535]<->NULL
MAIN[66536:82919]->P[66536:82915]<->H[82916:82919]<->NULL
MAIN[82920:99303]->P[82920:99299]<->H[99300:99303]<->NULL
MAIN[99304:115687]->P[99304:115683]<->H[115684:115687]<->NULL
MAIN[115688:132071]->P[115688:132067]<->H[132068:132071]<->NULL
MAIN[132072:148455]->P[132072:148451]<->H[148452:148455]<->NULL
MAIN[148456:164839]->P[148456:164835]<->H[164836:164839]<->NULL
MAIN[164840:181223]->P[164840:181219]<->H[181220:181223]<->NULL
MAIN[181224:197607]->P[181224:197603]<->H[197604:197607]<->NULL
MAIN[197608:213991]->P[197608:213987]<->H[213988:213991]<->NULL
MAIN[213992:230375]->P[213992:230371]<->H[230372:230375]<->NULL
MAIN[230376:246759]->P[230376:246755]<->H[246756:246759]<->NULL
MAIN[246760:263143]->P[246760:263139]<->H[263140:263143]<->NULL
MAIN[263144:279527]->P[263144:279523]<->H[279524:279527]<->NULL
MAIN[279528:295911]->P[279528:295907]<->H[295908:295911]<->NULL
MAIN[295912:312295]->P[295912:312291]<->H[312292:312295]<->NULL
MAIN[312296:328679]->P[312296:328675]<->H[328676:328679]<->NULL
MAIN[328680:345063]->P[328680:345059]<->H[345060:345063]<->NULL
MAIN[345064:361447]->P[345064:361443]<->H[361444:361447]<->NULL
MAIN[361448:377831]->P[361448:377827]<->H[377828:377831]<->NULL
MAIN[377832:394215]->P[377832:394211]<->H[394212:394215]<->NULL
MAIN[394216:410599]->P[394216:410595]<->H[410596:410599]<->NULL
MAIN[410600:426983]->P[410600:426979]<->H[426980:426983]<->NULL
MAIN[426984:443367]->P[426984:443363]<->H[443364:443367]<->NULL
MAIN[443368:459751]->P[443368:459747]<->H[459748:459751]<->NULL
MAIN[459752:476135]->P[459752:476131]<->H[476132:476135]<->NULL
MAIN[476136:492519]->P[476136:492515]<->H[492516:492519]<->NULL
MAIN[492520:508903]->P[492520:508899]<->H[508900:508903]<->NULL
MAIN[508904:525287]->P[508904:525283]<->H[525284:525287]<->NULL
MAIN[525288:541671]->P[525288:541667]<->H[541668:541671]<->NULL
MAIN[541672:558055]->P[541672:558051]<->H[558052:558055]<->NULL
MAIN[558056:574439]->P[558056:574435]<->H[574436:574439]<->NULL
MAIN[574440:590823]->P[574440:590819]<->H[590820:590823]<->NULL
MAIN[590824:607207]->P[590824:607203]<->H[607204:607207]<->NULL
MAIN[607208:623591]->P[607208:623587]<->H[623588:623591]<->NULL
MAIN[623592:639975]->P[623592:639971]<->H[639972:639975]<->NULL
MAIN[639976:656359]->P[639976:656355]<->H[656356:656359]<->NULL
MAIN[656360:672743]->P[656360:672739]<->H[672740:672743]<->NULL
MAIN[672744:689127]->P[672744:689123]<->H[689124:689127]<->NULL
MAIN[689128:705511]->P[689128:705507]<->H[705508:705511]<->NULL
MAIN[705512:721895]->P[705512:721891]<->H[721892:721895]<->NULL
MAIN[721896:738279]->P[721896:738275]<->H[738276:738279]<->NULL
MAIN[738280:754663]->P[738280:754659]<->H[754660:754663]<->NULL
MAIN[754664:771047]->P[754664:771043]<->H[771044:771047]<->NULL
MAIN[771048:787431]->P[771048:787427]<->H[787428:787431]<->NULL
MAIN[787432:803815]->P[787432:803811]<->H[803812:803815]<->NULL
MAIN[803816:820199]->P[803816:820195]<->H[820196:820199]<->NULL
```

```
MAIN[820200:836583]->P[820200:836579]<->H[836580:836583]<->NULL
MAIN[836584:852967]->P[836584:852963]<->H[852964:852967]<->NULL
MAIN[852968:869351]->P[852968:869347]<->H[869348:869351]<->NULL
MAIN[869352:885735]->P[869352:885731]<->H[885732:885735]<->NULL
MAIN[885736:902119]->P[885736:902115]<->H[902116:902119]<->NULL
MAIN[902120:918503]->P[902120:918499]<->H[918500:918503]<->NULL
MAIN[918504:934887]->P[918504:934883]<->H[934884:934887]<->NULL
MAIN[934888:951271]->P[934888:951267]<->H[951268:951271]<->NULL
MAIN[951272:967655]->P[951272:967651]<->H[967652:967655]<->NULL
MAIN[967656:984039]->P[967656:984035]<->H[984036:984039]<->NULL
MAIN[984040:1000423]->P[984040:1000419]<->H[1000420:1000423]<->NULL
MAIN[1000424:1016807]->P[1000424:1016803]<->H[1016804:1016807]<->NULL
MAIN[1016808:1033191]->P[1016808:1033187]<->H[1033188:1033191]<->NULL
MAIN[1033192:1049575]->P[1033192:1049571]<->H[1049572:1049575]<->NULL
MAIN[1049576:1065959]->P[1049576:1065955]<->H[1065956:1065959]<->NULL
MAIN[1065960:1082343]->P[1065960:1082339]<->H[1082340:1082343]<->NULL
MAIN[1082344:1098727]->P[1082344:1098723]<->H[1098724:1098727]<->NULL
MAIN[1098728:1115111]->P[1098728:1115107]<->H[1115108:1115111]<->NULL
MAIN[1115112:1131495]->P[1115112:1131491]<->H[1131492:1131495]<->NULL
MAIN[1131496:1147879]->P[1131496:1147875]<->H[1147876:1147879]<->NULL
MAIN[1147880:1164263]->P[1147880:1164259]<->H[1164260:1164263]<->NULL
MAIN[1164264:1180647]->P[1164264:1180643]<->H[1180644:1180647]<->NULL
MAIN[1180648:1197031]->P[1180648:1197027]<->H[1197028:1197031]<->NULL
MAIN[1197032:1213415]->P[1197032:1213411]<->H[1213412:1213415]<->NULL
Pages used: 296
Space unused:  296
Main Chain Length:     74
Sub-Chain Length array: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, ]
----------------------------------------

--------- Freeing up the memory [mems_free] --------
---------MeMS SYSTEM STATS------------
MAIN[1000:17383]->P[1000:17379]<->H[17380:17383]<->NULL
MAIN[17384:33767]->P[17384:33763]<->H[33764:33767]<->NULL
MAIN[33768:50151]->P[33768:50147]<->H[50148:50151]<->NULL
MAIN[50152:66535]->H[50152:66535]<->NULL
MAIN[66536:82919]->P[66536:82915]<->H[82916:82919]<->NULL
MAIN[82920:99303]->P[82920:99299]<->H[99300:99303]<->NULL
MAIN[99304:115687]->P[99304:115683]<->H[115684:115687]<->NULL
MAIN[115688:132071]->P[115688:132067]<->H[132068:132071]<->NULL
MAIN[132072:148455]->P[132072:148451]<->H[148452:148455]<->NULL
MAIN[148456:164839]->P[148456:164835]<->H[164836:164839]<->NULL
MAIN[164840:181223]->P[164840:181219]<->H[181220:181223]<->NULL
MAIN[181224:197607]->P[181224:197603]<->H[197604:197607]<->NULL
```

```
MAIN[197608:213991]->P[197608:213987]<->H[213988:213991]<->NULL
MAIN[213992:230375]->P[213992:230371]<->H[230372:230375]<->NULL
MAIN[230376:246759]->P[230376:246755]<->H[246756:246759]<->NULL
MAIN[246760:263143]->P[246760:263139]<->H[263140:263143]<->NULL
MAIN[263144:279527]->P[263144:279523]<->H[279524:279527]<->NULL
MAIN[279528:295911]->P[279528:295907]<->H[295908:295911]<->NULL
MAIN[295912:312295]->P[295912:312291]<->H[312292:312295]<->NULL
MAIN[312296:328679]->P[312296:328675]<->H[328676:328679]<->NULL
MAIN[328680:345063]->P[328680:345059]<->H[345060:345063]<->NULL
MAIN[345064:361447]->P[345064:361443]<->H[361444:361447]<->NULL
MAIN[361448:377831]->P[361448:377827]<->H[377828:377831]<->NULL
MAIN[377832:394215]->P[377832:394211]<->H[394212:394215]<->NULL
MAIN[394216:410599]->P[394216:410595]<->H[410596:410599]<->NULL
MAIN[410600:426983]->P[410600:426979]<->H[426980:426983]<->NULL
MAIN[426984:443367]->P[426984:443363]<->H[443364:443367]<->NULL
MAIN[443368:459751]->P[443368:459747]<->H[459748:459751]<->NULL
MAIN[459752:476135]->P[459752:476131]<->H[476132:476135]<->NULL
MAIN[476136:492519]->P[476136:492515]<->H[492516:492519]<->NULL
MAIN[492520:508903]->P[492520:508899]<->H[508900:508903]<->NULL
MAIN[508904:525287]->P[508904:525283]<->H[525284:525287]<->NULL
MAIN[525288:541671]->P[525288:541667]<->H[541668:541671]<->NULL
MAIN[541672:558055]->P[541672:558051]<->H[558052:558055]<->NULL
MAIN[558056:574439]->P[558056:574435]<->H[574436:574439]<->NULL
MAIN[574440:590823]->P[574440:590819]<->H[590820:590823]<->NULL
MAIN[590824:607207]->P[590824:607203]<->H[607204:607207]<->NULL
MAIN[607208:623591]->P[607208:623587]<->H[623588:623591]<->NULL
MAIN[623592:639975]->P[623592:639971]<->H[639972:639975]<->NULL
MAIN[639976:656359]->P[639976:656355]<->H[656356:656359]<->NULL
MAIN[656360:672743]->P[656360:672739]<->H[672740:672743]<->NULL
MAIN[672744:689127]->P[672744:689123]<->H[689124:689127]<->NULL
MAIN[689128:705511]->P[689128:705507]<->H[705508:705511]<->NULL
MAIN[705512:721895]->P[705512:721891]<->H[721892:721895]<->NULL
MAIN[721896:738279]->P[721896:738275]<->H[738276:738279]<->NULL
MAIN[738280:754663]->P[738280:754659]<->H[754660:754663]<->NULL
MAIN[754664:771047]->P[754664:771043]<->H[771044:771047]<->NULL
MAIN[771048:787431]->P[771048:787427]<->H[787428:787431]<->NULL
MAIN[787432:803815]->P[787432:803811]<->H[803812:803815]<->NULL
MAIN[803816:820199]->P[803816:820195]<->H[820196:820199]<->NULL
MAIN[820200:836583]->P[820200:836579]<->H[836580:836583]<->NULL
MAIN[836584:852967]->P[836584:852963]<->H[852964:852967]<->NULL
MAIN[852968:869351]->P[852968:869347]<->H[869348:869351]<->NULL
MAIN[869352:885735]->P[869352:885731]<->H[885732:885735]<->NULL
MAIN[885736:902119]->P[885736:902115]<->H[902116:902119]<->NULL
MAIN[902120:918503]->P[902120:918499]<->H[918500:918503]<->NULL
MAIN[918504:934887]->P[918504:934883]<->H[934884:934887]<->NULL
MAIN[934888:951271]->P[934888:951267]<->H[951268:951271]<->NULL
MAIN[951272:967655]->P[951272:967651]<->H[967652:967655]<->NULL
```

```
MAIN[967656:984039]->P[967656:984035]<->H[984036:984039]<->NULL
MAIN[984040:1000423]->P[984040:1000419]<->H[1000420:1000423]<->NULL
MAIN[1000424:1016807]->P[1000424:1016803]<->H[1016804:1016807]<->NULL
MAIN[1016808:1033191]->P[1016808:1033187]<->H[1033188:1033191]<->NULL
MAIN[1033192:1049575]->P[1033192:1049571]<->H[1049572:1049575]<->NULL
MAIN[1049576:1065959]->P[1049576:1065955]<->H[1065956:1065959]<->NULL
MAIN[1065960:1082343]->P[1065960:1082339]<->H[1082340:1082343]<->NULL
MAIN[1082344:1098727]->P[1082344:1098723]<->H[1098724:1098727]<->NULL
MAIN[1098728:1115111]->P[1098728:1115107]<->H[1115108:1115111]<->NULL
MAIN[1115112:1131495]->P[1115112:1131491]<->H[1131492:1131495]<->NULL
MAIN[1131496:1147879]->P[1131496:1147875]<->H[1147876:1147879]<->NULL
MAIN[1147880:1164263]->P[1147880:1164259]<->H[1164260:1164263]<->NULL
MAIN[1164264:1180647]->P[1164264:1180643]<->H[1180644:1180647]<->NULL
MAIN[1180648:1197031]->P[1180648:1197027]<->H[1197028:1197031]<->NULL
MAIN[1197032:1213415]->P[1197032:1213411]<->H[1213412:1213415]<->NULL
Pages used: 296
Space unused:  16676
Main Chain Length:     74
Sub-Chain Length array: [2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, ]
----------------------------------------
---------MeMS SYSTEM STATS------------
MAIN[1000:17383]->P[1000:17379]<->H[17380:17383]<->NULL
MAIN[17384:33767]->P[17384:33763]<->H[33764:33767]<->NULL
MAIN[33768:50151]->P[33768:50147]<->H[50148:50151]<->NULL
MAIN[50152:66535]->P[50152:51151]<->H[51152:66535]<->NULL
MAIN[66536:82919]->P[66536:82915]<->H[82916:82919]<->NULL
MAIN[82920:99303]->P[82920:99299]<->H[99300:99303]<->NULL
MAIN[99304:115687]->P[99304:115683]<->H[115684:115687]<->NULL
MAIN[115688:132071]->P[115688:132067]<->H[132068:132071]<->NULL
MAIN[132072:148455]->P[132072:148451]<->H[148452:148455]<->NULL
MAIN[148456:164839]->P[148456:164835]<->H[164836:164839]<->NULL
MAIN[164840:181223]->P[164840:181219]<->H[181220:181223]<->NULL
MAIN[181224:197607]->P[181224:197603]<->H[197604:197607]<->NULL
MAIN[197608:213991]->P[197608:213987]<->H[213988:213991]<->NULL
MAIN[213992:230375]->P[213992:230371]<->H[230372:230375]<->NULL
MAIN[230376:246759]->P[230376:246755]<->H[246756:246759]<->NULL
MAIN[246760:263143]->P[246760:263139]<->H[263140:263143]<->NULL
MAIN[263144:279527]->P[263144:279523]<->H[279524:279527]<->NULL
MAIN[279528:295911]->P[279528:295907]<->H[295908:295911]<->NULL
MAIN[295912:312295]->P[295912:312291]<->H[312292:312295]<->NULL
MAIN[312296:328679]->P[312296:328675]<->H[328676:328679]<->NULL
MAIN[328680:345063]->P[328680:345059]<->H[345060:345063]<->NULL
MAIN[345064:361447]->P[345064:361443]<->H[361444:361447]<->NULL
MAIN[361448:377831]->P[361448:377827]<->H[377828:377831]<->NULL
```

```
MAIN[377832:394215]->P[377832:394211]<->H[394212:394215]<->NULL
MAIN[394216:410599]->P[394216:410595]<->H[410596:410599]<->NULL
MAIN[410600:426983]->P[410600:426979]<->H[426980:426983]<->NULL
MAIN[426984:443367]->P[426984:443363]<->H[443364:443367]<->NULL
MAIN[443368:459751]->P[443368:459747]<->H[459748:459751]<->NULL
MAIN[459752:476135]->P[459752:476131]<->H[476132:476135]<->NULL
MAIN[476136:492519]->P[476136:492515]<->H[492516:492519]<->NULL
MAIN[492520:508903]->P[492520:508899]<->H[508900:508903]<->NULL
MAIN[508904:525287]->P[508904:525283]<->H[525284:525287]<->NULL
MAIN[525288:541671]->P[525288:541667]<->H[541668:541671]<->NULL
MAIN[541672:558055]->P[541672:558051]<->H[558052:558055]<->NULL
MAIN[558056:574439]->P[558056:574435]<->H[574436:574439]<->NULL
MAIN[574440:590823]->P[574440:590819]<->H[590820:590823]<->NULL
MAIN[590824:607207]->P[590824:607203]<->H[607204:607207]<->NULL
MAIN[607208:623591]->P[607208:623587]<->H[623588:623591]<->NULL
MAIN[623592:639975]->P[623592:639971]<->H[639972:639975]<->NULL
MAIN[639976:656359]->P[639976:656355]<->H[656356:656359]<->NULL
MAIN[656360:672743]->P[656360:672739]<->H[672740:672743]<->NULL
MAIN[672744:689127]->P[672744:689123]<->H[689124:689127]<->NULL
MAIN[689128:705511]->P[689128:705507]<->H[705508:705511]<->NULL
MAIN[705512:721895]->P[705512:721891]<->H[721892:721895]<->NULL
MAIN[721896:738279]->P[721896:738275]<->H[738276:738279]<->NULL
MAIN[738280:754663]->P[738280:754659]<->H[754660:754663]<->NULL
MAIN[754664:771047]->P[754664:771043]<->H[771044:771047]<->NULL
MAIN[771048:787431]->P[771048:787427]<->H[787428:787431]<->NULL
MAIN[787432:803815]->P[787432:803811]<->H[803812:803815]<->NULL
MAIN[803816:820199]->P[803816:820195]<->H[820196:820199]<->NULL
MAIN[820200:836583]->P[820200:836579]<->H[836580:836583]<->NULL
MAIN[836584:852967]->P[836584:852963]<->H[852964:852967]<->NULL
MAIN[852968:869351]->P[852968:869347]<->H[869348:869351]<->NULL
MAIN[869352:885735]->P[869352:885731]<->H[885732:885735]<->NULL
MAIN[885736:902119]->P[885736:902115]<->H[902116:902119]<->NULL
MAIN[902120:918503]->P[902120:918499]<->H[918500:918503]<->NULL
MAIN[918504:934887]->P[918504:934883]<->H[934884:934887]<->NULL
MAIN[934888:951271]->P[934888:951267]<->H[951268:951271]<->NULL
MAIN[951272:967655]->P[951272:967651]<->H[967652:967655]<->NULL
MAIN[967656:984039]->P[967656:984035]<->H[984036:984039]<->NULL
MAIN[984040:1000423]->P[984040:1000419]<->H[1000420:1000423]<->NULL
MAIN[1000424:1016807]->P[1000424:1016803]<->H[1016804:1016807]<->NULL
MAIN[1016808:1033191]->P[1016808:1033187]<->H[1033188:1033191]<->NULL
MAIN[1033192:1049575]->P[1033192:1049571]<->H[1049572:1049575]<->NULL
MAIN[1049576:1065959]->P[1049576:1065955]<->H[1065956:1065959]<->NULL
MAIN[1065960:1082343]->P[1065960:1082339]<->H[1082340:1082343]<->NULL
MAIN[1082344:1098727]->P[1082344:1098723]<->H[1098724:1098727]<->NULL
MAIN[1098728:1115111]->P[1098728:1115107]<->H[1115108:1115111]<->NULL
MAIN[1115112:1131495]->P[1115112:1131491]<->H[1131492:1131495]<->NULL
MAIN[1131496:1147879]->P[1131496:1147875]<->H[1147876:1147879]<->NULL
```

```
MAIN[1147880:1164263]->P[1147880:1164259]<->H[1164260:1164263]<->NULL
MAIN[1164264:1180647]->P[1164264:1180643]<->H[1180644:1180647]<->NULL
MAIN[1180648:1197031]->P[1180648:1197027]<->H[1197028:1197031]<->NULL
MAIN[1197032:1213415]->P[1197032:1213411]<->H[1213412:1213415]<->NULL
Pages used: 296
Space unused:  15676
Main Chain Length:     74
Sub-Chain Length array: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, ]
----------------------------------------

--------- Unmapping all memory [mems_finish] --------
```

The mmap is still done after the first mmap to fill the nodes is exhausted
.

Test - 3

Input

```
// include other header files as needed
#include"mems.h"


int main(int argc, char const *argv[])
{
   // initialise the MeMS system
   mems_init();
   int* ptr[10];

   /*
   This allocates 10 arrays of 250 integers each
   */
   printf("\n------- Allocated virtual addresses [mems_malloc]
-------\n");
   for(int i=0;i<10;i++){
       ptr[i] = (int*)mems_malloc(sizeof(int)*250);
       printf("Virtual address: %lu\n", (unsigned long)ptr[i]);
   }

   /*
   In this section we are tring to write value to 1st index of array[0]
(here it is 0 based indexing).
```

```
    We get get value of both the 0th index and 1st index of array[0] by
using function mems_get.
    Then we write value to 1st index using 1st index pointer and try to
access it via 0th index pointer.

    This section is show that even if we have allocated an array using
mems_malloc but we can
    retrive MeMS physical address of any of the element from that array
using mems_get.
    */
    printf("\n------ Assigning value to Virtual address [mems_get]
-----\n");
    // how to write to the virtual address of the MeMS (this is given to
show that the system works on arrays as well)
    int* phy_ptr= (int*) mems_get(&ptr[0][1]); // get the address of index
1
    phy_ptr[0]=200; // put value at index 1
    int* phy_ptr2= (int*) mems_get(&ptr[0][0]); // get the address of index
0
    printf("Virtual address: %lu\tPhysical Address: %lu\n",(unsigned
long)ptr[0],(unsigned long)phy_ptr2);
    printf("Value written: %d\n", phy_ptr2[1]); // print the address of
index 1

    /*
    This shows the stats of the MeMS system.
    */
    printf("\n--------- Printing Stats [mems_print_stats] --------\n");
    mems_print_stats();

    /*
    This section shows the effect of freeing up space on free list and also
the effect of
    reallocating the space that will be fullfilled by the free list.
    */
    printf("\n--------- Freeing up the memory [mems_free] --------\n");
    mems_free(ptr[6]);
    mems_free(ptr[7]);
    mems_print_stats();
    ptr[3] = (int*)mems_malloc(sizeof(int)*250);
```

```
    mems_print_stats();

    printf("\n--------- Unmapping all memory [mems_finish] --------\n\n");
    mems_finish();
    return 0;
}
```

Output

```
------- Allocated virtual addresses [mems_malloc] -------
Virtual address: 1000
Virtual address: 2000
Virtual address: 3000
Virtual address: 4000
Virtual address: 5096
Virtual address: 6096
Virtual address: 7096
Virtual address: 8096
Virtual address: 9192
Virtual address: 10192

------ Assigning value to Virtual address [mems_get] -----
Virtual address: 1000   Physical Address: 139762912509952
Value written: 200

--------- Printing Stats [mems_print_stats] --------
---------MeMS SYSTEM STATS------------
MAIN[1000:5095]->P[1000:1999]<->P[2000:2999]<->P[3000:3999]<->P[4000:4999]
<->H[5000:5095]<->NULL
MAIN[5096:9191]->P[5096:6095]<->P[6096:7095]<->P[7096:8095]<->P[8096:9095]
<->H[9096:9191]<->NULL
MAIN[9192:13287]->P[9192:10191]<->P[10192:11191]<->H[11192:13287]<->NULL
Pages used: 3
Space unused:  2288
Main Chain Length:     3
Sub-Chain Length array: [5, 5, 3, ]
----------------------------------------

--------- Freeing up the memory [mems_free] --------
---------MeMS SYSTEM STATS------------
MAIN[1000:5095]->P[1000:1999]<->P[2000:2999]<->P[3000:3999]<->P[4000:4999]
<->H[5000:5095]<->NULL
MAIN[5096:9191]->P[5096:6095]<->P[6096:7095]<->H[7096:9191]<->NULL
MAIN[9192:13287]->P[9192:10191]<->P[10192:11191]<->H[11192:13287]<->NULL
Pages used: 3
Space unused:  4288
Main Chain Length:     3
```

```
Sub-Chain Length array: [5, 3, 3, ]
----------------------------------------
---------MeMS SYSTEM STATS------------
MAIN[1000:5095]->P[1000:1999]<->P[2000:2999]<->P[3000:3999]<->P[4000:4999]
<->H[5000:5095]<->NULL
MAIN[5096:9191]->P[5096:6095]<->P[6096:7095]<->P[7096:8095]<->H[8096:9191]
<->NULL
MAIN[9192:13287]->P[9192:10191]<->P[10192:11191]<->H[11192:13287]<->NULL
Pages used: 3
Space unused:  3288
Main Chain Length:     3
Sub-Chain Length array: [5, 4, 3, ]
----------------------------------------

--------- Unmapping all memory [mems_finish] --------
```

In the output of the first print_stats after freeing ptr[6] and ptr[7] , we can see that the three consecutive holes have combined to give one single hole .

Error Handling :

On each mmap we are doing if map is failed it gives -1 as value and it will give error .

Some examples are :

```
   vir_address = 1000 ;
   head = (main_node*)mmap(NULL, PAGE_SIZE*1,
PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE , -1, 0);
   if(head == MAP_FAILED){
       perror("mmap failed");
   }
```

```
  else{
     //printf("here\n") ;
       node = (main_node*)mmap(NULL, PAGE_SIZE*1,
PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE ,-1,0) ;
       if(node == MAP_FAILED){
           perror("mmap failed");
       }
       chain_count++ ;
else {
       //printf("here\n") ;
```

```
       new_sub =(subchain_node*
)mmap(NULL,PAGE_SIZE*1,PROT_READ|PROT_WRITE,MAP_ANONYMOUS|MAP_PRIVATE
,-1,0) ;
       if(new_sub == MAP_FAILED){
           perror("mmap failed");
       }
       subchain_count_ls++ ;
      // printf("%d\n",subchain_count_ls);
    init_sub = new_sub ;
       current_pointer_subchain =
```

And so on . . . . .

With this our documentation comes to the end .