A "BETTER" MALLOC

- A Project By:
"Team Uninitialized"
Mitesh Shah (20172010)
Rahil Sheth (20172062)
Sweta Kumari (20172126)
Neha Gupta (20172130)

PROBLEM STATEMENT

The basic idea of this project was to understand how a simple memory allocator works and then write one of our own using basic system calls.

The implementation must pass through some basic guidelines:

(1) To request memory from OS, we must use *mmap()* system call instead of *sbrk().* (which is usually used by the normal memory allocators).
(2) Originally a memory allocator requests more memory from OS whenever it cant satisfy any request for which it uses *sbrk()* system call. Our allocator should call *mmap()* only once (when it is first initialized)
(3) The choice for Data Structures to maintain the free list and policy for choosing the chunk of memory was upto us.
(4) **The memory allocator should be more flexible in how the user can specify what memory should be freed.**

In the normal implementation of **malloc(),** this is how **free()** works:

i) **Free(void *ptr)** frees the memory space pointed by the pointer ptr which must be returned by a previous call to malloc() or calloc() or realloc(), otherwise if free(ptr) has already been called before, then undefined behavior occurs.
ii) If ptr is NULL, no operation is performed.
iii) When we call free(ptr), the ptr must point to the beginning of the memory block assigned.
iv) Whenever free(ptr) is called, the whole block of memory that was allocated is free i.e. you cannot de-allocate a part of memory!

Our implementation of **Mem_Alloc(int size)** and **Mem_Free(void *ptr)** are identical except the following differences:

i) The pointer passed to Mem_Free does not have to have been previously returned by Mem_Alloc(). Instead, ptr can point to any valid range of memory returned by Mem_Alloc(). **Example:** Call to Mem_Free can be of this type:

```
int *ptr;
```

```
if((ptr==(int *)Mem_Alloc(sizeof(int)*50))==NULL) exit(1);
Mem_Free(ptr+30);
```

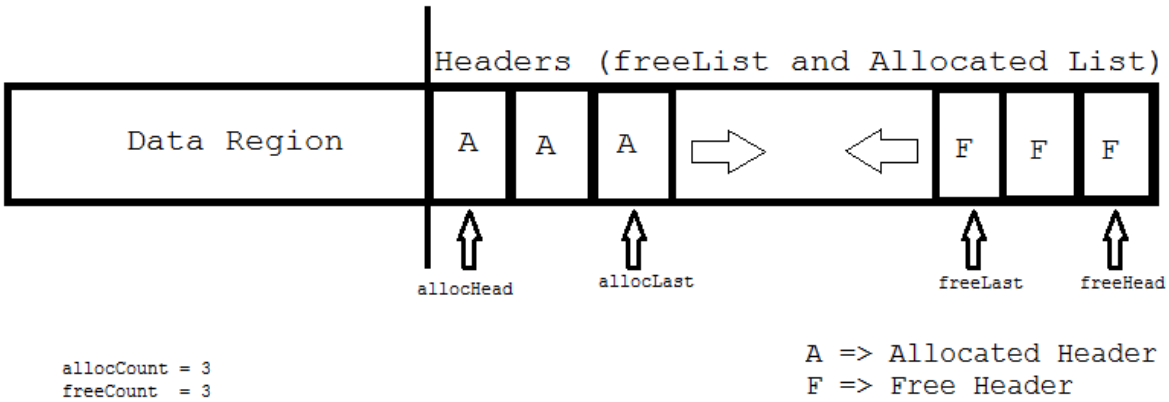
This 30 could be replaced by any value and the block must be freed from there till the end.

Thus we must design a sophisticated data structure than the traditional malloc to track the regions of memory allocated by Mem_Alloc.

How We Tackle The Problem?

We wrote our own allocator in “mem.cpp” and declare the functions “mem.h” that can be used directly in any program by including the library. To implement the required functions viz. **Mem_Alloc(), Mem_Free(), Mem_isValid()** and **Mem_getSize(),** we follow the following strategy and define the following functions and also how the free list and allocated list are maintained:

How Our Data Structure Works?



As we request 4 times the memory the user requests, we use one portion of it to store and allocate the blocks that the user requests. For each block that has been allocated or deallocated, a new header is created in the respective free list and allocated list that are dynamically created in the remaining portion of the requested memory.

Both the lists grow in opposite directions as depicted in the diagram above. Also the variables ***allocHead & allocLast*** are used to maintain the head and tail of the allocated list, and ***freeHead & freeLast*** are used to maintain the head and tail of the allocated list. Since we have enough contiguous memory on the both sides, rather than creating and storing a pointer to the next header in our structure, we can access it in a sequential fashion directly (like an array!).

0 : GLOBAL VARIABLES AND DATA STRUCTURES :

We define the following data structures and variables:

(1) `void *ptr = NULL` : Just a temporary pointer to use in various functions.
(2) The Data Structure to store headers corresponding to the memory blocks assigned to the user:

```
struct mem_node {
  void *data; //This points to the base address of the data block assigned
  int size;    //This is the size of the data block to this corresponding header.

  }
```

(3) `struct mem_node *freeHead` : This is the pointer to the start of the list of headers representing the blocks of memory that are free.
(4) `struct mem_node *allocHead` : This is the pointer to the start of the list of headers representing the blocks of memory that are allocated to the user.
(5) `int freeCount = 1` : This represents the count of free blocks. This is initialized with 1 because when Mem_Init() is called initially, there is a single free block.
(6) `int allocCount = 0` : This represents the count of blocks that are allocated to the user. This is initialized with 0 because initially no blocks are allocated to the user.
(7) `int presentspace` : Used to keep track of current free space.
(8) `mem_node *freeLast = NULL` : This is the pointer to the last element of list of free headers.
(9) `mem_node *allocLast = NULL` : This is the pointer to the last element of list of allocated headers.

The first routine we define is **Mem_Init** and this is how it works:

1 : DEFINITION OF MEM_INIT()

`int Mem_Init(int sizeOfRegion) :`

Mem_Init is called one time by user program that is going to use our routines. sizeOfRegion is the maximum total number of bytes that the process can request and this memory is requested from the OS by us using **mmap().**

Here is how it is implemented in our code:

1) We check if the provided **sizeOfRegion** is valid or not. If it is invalid (negative), then we return -1.

```
if(sizeOfRegion<=0)
```

```
{
      perror("invalid size of region");
      return -1;
}
```

2) Here we are requesting memory from OS using **mmap()** . Here we are requesting **4*sizeOfRegion** i.e. four times the number of bytes requested by the user to store our headers for the free and allocated blocks.

```
int fd = open("/dev/zero", O_RDWR);
ptr = mmap(NULL, 4*sizeOfRegion, PROT_READ | PROT_WRITE,
MAP_PRIVATE, fd, 0);
if (ptr == MAP_FAILED) {
perror("mmap");
return -1;
}
```

3) We store the value of current free space available in ***presentspace.*** Also we initialize ***freeHead*** to point to the end of the memory region and initialize ***allocHead*** after the data region where we will store our headers. We also store the address of beginning of block in the data pointer of ***freeHead*** since that is the current free block and size member as sizeOfRegion. Similarly we initialize ***freelast, alloclast, sizeOfRegion (****Global Variable****)*** .

```
presentspace=sizeOfRegion;
freeHead=(mem_node*)(ptr + 4*sizeOfRegion - sizeof(mem_node));
allocHead=(mem_node*)(ptr + sizeOfRegion);
allocHead->size=0;
freeHead->data=ptr;
freeHead->size=sizeOfRegion;
freelast=freeHead;
alloclast=allocHead;
close(fd);
return 1;
```

If **Mem_Init()** is successful in obtaining a block, it returns 1, otherwise it returns 0.

2 : DEFINITION OF MEM_ALLOC()

`void* Mem_Alloc(int size) :`

It takes the size of bytes to be allocated as input and returns a void pointer to the start to the allocated block of memory. Function returns NULL if no free space is available to satisfy the request.

We are using **WORST FIT** to find the free block in the memory and allocate it to the user (as shown in step 2)). First we check the available present space (using the ***presentspace***) variable and if the requested size is less than its value, then we move ahead. We find the biggest free available block in our region by traversing the free list completely (Step 2)). Then we handle 3 cases:

First Case: is when user requests an amount of memory that is less than the size of the biggest available free block (node) in the region (Check Step 3)):

> In this case, we reduce the size of the biggest node by the amount of size requested by the user and then create a header in the allocated list for that respective allocated block and initialize it with proper values. We also reduce the value of ***presentspace*** accordingly.

Second Case: is when user requests memory equal to the size of biggest available node (Check Step 4)):

> Here we just create a header in the allocated list for that respective allocated block and initialize it with proper values.
>
> Then we check if the biggest node was the last free node in our free list. If it wasn't, we initialize it accordingly (i.e. we move the **freelast** pointer accordingly and decrement the value of **freecount**). If the biggest node was the last free node in our free list then we assign NULL to the **freeHead** pointer because there is no free blocks available now.

Third Case: is when the user requests more memory than the current size of biggest node. In this case, it may so happen that the ***presentspace*** variable shows more size available but the biggest node's size is less than the requested size, because even after merging adjacent free blocks, there are some non-contiguous free blocks that can't be merged and thus the request of user can't be processed (Check Step 5)).

> In this case we display "Not Enough Memory" and return a NULL pointer.

In first and second case, we also check if any headers were created earlier. If there are any preexisting headers then we append the new header and if there isn't any preexisting header then we create a header and initialize ***allocHead.***

1) We begin with checking if there is any space available in our region to allocate, so we compare it the asked size with the ***presentspace.***

```
if(size>presentspace)
{
      cout<<"not enough memory\n";
```

```
        return NULL;
}
```

2) Now we find the biggest available node in our region of memory (i.e. implementation of Worst Fit algorithm) by traversing the free list (using ***freeHead and freeCount***) variables and set the value of ***biggestnode*** pointer. Here we also create a pointer ***userptr*** that will be initialized with the data region pointed by the ***biggestnode*** and eventually be returned to the user.

```
mem_node *biggestnode=freeHead;
mem_node *currentnode=biggestnode;
void *userptr=biggestnode->data;
int biggestsize=biggestnode->size;
for(int i=1;i<=freecount;i++)
{
        if(biggestnode->size < currentnode->size)
        {
                biggestnode=currentnode;
        }
        currentnode--;
}
userptr=biggestnode->data;
mem_node *newallo;
```

3) Now we handle the First Case as stated above (i.e. check if the size of biggest node is less than the requested size), decrease the size of biggest head and then we create a new header. We also check the case when there are no headers created (the "else" part where "newallo=allochHead" is done). Finally, we decrease the size of present space variable accordingly.

```
if(size<biggestnode->size)
{
        biggestnode->size-=size;
        biggestnode->data+=size;
        if(allocHead->size != 0)
```

```
        {
                newallo=alloclast +1;
        }
        else
                newallo=allocHead;
        newallo->size=size;
        newallo->data=userptr;
        alloccount++;
        alloclast=newallo;
        presentspace-=size;
}
```

4) Now we handle the Second Case as stated above (i.e. if the size of biggest node is equal to the requested size). Here we create a new allocated header and thus increment the no of allocated headers and correspondingly,remove the biggest node from the free block headers and thus decrementing the no of freeblocks, accordingly. Then we check if the biggest node was the last node or not and take proper steps (as mentioned above).

```
else if(size==biggestnode->size)
{
        if(allocHead->size != 0)
                newallo=alloclast + 1;
        else
                newallo=allocHead;
        newallo->size=size;
        newallo->data=userptr;
        alloccount++;
        alloclast=newallo;
        presentspace-=size;
        if(biggestnode!=freelast)
        {
                biggestnode->data=freelast->data;
```

```
            biggestnode->size=freelast->size;
      }
      freelast++;
      if(freecount>1)
            freecount--;
      else if(freecount==1)
      {
            freeHead->data=NULL;
            freeHead->size=0;
      }
}
```

5) Now we handle the Third Case as stated above (i.e. if the size of biggest node is less than the requested size). Then we won't be able to allocate the memory to the user and thus display "Not Enough Memory" and return a NULL Pointer.

```
else
{
      cout<<"not enough memory\n";
      return NULL;
}
```

6) At the end, we return the assigned value of ***userptr*** to the user.

```
return userptr;
```

3 : DEFINITION OF MEM_FREE()

`int Mem_Free(void *ptr1):`

Mem_Free() frees the memory object that **ptr1** falls within, according to the rules described in **Problem Statement.** Just like standard free if **ptr1** is NULL, then no operation is performed. The function returns 0 on success and -1 if **ptr1** does not fall within a currently allocated object.

Following cases are handled:

First Case: is when the value of **ptr1** is NULL (Check Step 1)):

In this case, we return -1 as it isn't valid request for deallocation.

Second Case: Now we pass the value of **ptr1** to **Mem_isValid()** to check the validity of **ptr1** (check the implementation of **Mem_isValid()** to understand its use) (Check Step 1) and Step 2)).

If **Mem_isValid()** returns false, then its not a a valid pointer to free and thus we display the error and return -1.

If **ptr1** is certified valid by **Mem_isValid(),** then we traverse the whole list of allocated blocks and compare the data lower bound and upper bound of the data block to see if it lies in the range and find the node that contains **ptr1**. (Check Step 2))

Now the next step is to free the memory region specified by **ptr1**. There are two possible cases for the range specified by **ptr1.** We check this by checking the remaining size of currentnode (by subtracting the size of current node and the size of the region specified by **ptr1**).

Sub Case 1: when **ptr1** specifies a partial range (not the whole block)(Check Step 3)):

We create a header for the free list and also reduce the size for the header of the allocated list. And then we update **alloccount** and **freecount** accordingly.

Sub Case 2: when **ptr1** specifies the whole block to be free(Check Step 4)):

Here also we create a new header for the free list but we remove the header for the allocated block and update **alloccount** and **freecount** accordingly.

At the end we call the functions to merge adjacent nodes viz. ***mergeLeft*** *and* ***mergeRight*** (check their implementation for more details) and return 1(Check Step 5)).

1) Here we check if the value of **ptr1** is NULL or not**.** Then we check the validity of **ptr1** using **Mem_isValid()** function and return or move ahead accordingly.

```
if(ptr1==NULL)
      return -1;
if(!Mem_IsValid(ptr1))
{
      cout<<"Not a valid pointer\n";
      return -1;
}
```

2) Now we perform the necessary allocations and find the node in which **ptr1** lies and store it in **currentnode**.

```
mem_node *currentnode=allocHead;
unsigned long long int low,high;
for(int i=0;i<alloccount;i++)
{
      low=(uintptr_t)currentnode->data;
      high=low + currentnode->size;
      if((uintptr_t)low<=(uintptr_t)ptr1 &&
      (uintptr_t)ptr1<(uintptr_t)high)
      {
            break;
      }
      currentnode++;
}
int actual_size=currentnode->size;
currentnode->size=(uintptr_t)ptr1 - (uintptr_t)low;
```

3) Now according to Sub Case 1, we check if **ptr1** specifies a partial range. Then we perform the necessary steps as mentioned above.

```
if(currentnode->size!=1)
{

      mem_node *newfree=freelast;
      newfree--;
      newfree->size = actual_size - currentnode->size;
      newfree->data=ptr1;
      freecount++;
      freelast=newfree;
      presentspace+=newfree->size;
}
```

4) Now according to Sub Case 2, we check if **ptr1** specifies the whole block that is to be freed. Then we perform the necessary steps as mentioned above.

```
else
```

```
{
        mem_node *newfree=freelast - 1;
        newfree->size = actual_size;
        newfree->data=ptr1;
        freecount++;
        freelast=newfree;
        presentspace+=newfree->size;
        currentnode->size=alloclast->size;
        currentnode->data=alloclast->data;
        alloclast--;
        alloccount--;
}
```

5) Finally we call ***mergeLeft() and mergeRight()*** functions and return the necessary value.

```
mergeRight(newfree);
mergeLeft(newfree);
return 1;
```

4 : DEFINITION OF MEM_ISVALID()

`int Mem_isValid(void *ptr1):`

Mem_isValid() is used to check the validity of pointer passed to Mem_Free(). If the pointer fall in valid range it returns 1. Else it returns 0.

5 : DEFINITION OF MEM_GETSIZE()

`int Mem_GetSize(void *ptr1):`

It uses the same logic as **Mem_isValid()** and is just used for debugging purposes and to return the size of region if requested by user.

Then we define two merge functions **mergeLeft()** and **mergeRight()** which are used to merge the adjacent nodes.

6 : DEFINITION OF MERGERIGHT():

`void mergeRight(mem_node *ptr):`

In mergeRight() function, if there is a free node which can merged on the right side of the current node, when we traverse the whole free list, we merge them into a single node.

```
void mergeRight(mem_node *ptr)
{
      int low=(uintptr_t)ptr->data;
      int high=low + ptr->size;
      mem_node *currentnode=freeHead;
      int low1;
      for (int i = 0; i < freecount; ++i)
      {
            low1=(uintptr_t)currentnode->data;
            if(high == low1)
            {
                  ptr->size+=currentnode->size;
                  currentnode->size=freelast->size;
                  currentnode->data=freelast->data;
                  freelast++;
                  freecount--;
                  break;
            }
            currentnode--;
      }
      return;
}
```

7 : DEFINITION OF MERGELEFT()

mergeLeft() is implemented in the same way **mergeRight(),** i.e. we find a free node which can merged on the left side of the current node, when we traverse the whole free list, we merge them into a single node.

```
void mergeLeft(mem_node *ptr)
{
      int low=(uintptr_t)ptr->data;
      int high=low + ptr->size;
      mem_node *currentnode=freeHead;
      int high1;
      for (int i = 0; i < freecount; ++i)
      {
            high1=(uintptr_t)currentnode->data + currentnode
            ->size;
            if(high1 == low)
            {
                  currentnode->size+=ptr->size;
                  ptr->size=freelast->size;
                  ptr->data=freelast->data;
                  freelast++;
                  freecount--;
                  break;
            }
            currentnode--;
      }
      return;
```

```
}
```

CONCLUSION

Thus, we developed a simple memory allocator that mimics the working of malloc (not completely though) but lets us free memory in the way specified in the problem statement thus working “better” than the original malloc.