Operating Systems (OS)

A “BETTER” MALLOC

Project Report

By Team: ‘Bulbasaur’

Team Members:-

Varun Gupta – 2018201003
Shashi Jangra -- 2018202001
Anjul Ravi Gupta – 2018201021
Kishan Shankar Singhal – 2018201023

- **Project Objective:**

In this project, we are implementing a memory allocator for the heap of a user-level process. Our functions are similar to those provided by malloc() and free(), but a little more interesting. We are implementing following functionalities:-

1) **Malloc** - functions similar to traditional malloc, but **free** can free memory from any point in the allocated memory.

2) Implementations of **Mem_Alloc (int size)** and **Mem_Free (void *ptr)** are identical, except 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.

Memory allocators have two distinct tasks.

First, the memory allocator asks the operating system to expand the heap portion of the process's address space by calling either **sbrk** or **mmap**.

Second, the memory allocator doles out this memory to the calling process. This involves managing a free list of memory and finding a contiguous chunk of

memory that is large enough for the user's request; when the user later frees memory, it is added back to this list.

● Implementation:

The basic idea of this project was to understand how a simple memory allocator works using basic system calls.

The implementation must pass through some basic guidelines:

1) When requesting memory from the OS, we must use **mmap** (rather than sbrk).

2) Although a real memory allocator requests more memory from the OS whenever it can't satisfy a request from the user, our memory allocator must call **mmap only one time** (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.

The traditional **malloc()** and **free()** are defined as follows:-

- **void *malloc(size_t size)**:

  malloc() allocates **size** bytes and returns a pointer to the allocated memory. The memory is not cleared.

- **void free(void *ptr)**:

  free() frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc().

  Otherwise, or if free(ptr) has already been called before, undefined behaviour occurs.

  If ptr is NULL, no operation is performed.

Our implementations of **Mem_Alloc(int size)** and **Mem_Free(void *ptr)** are identical, except the **ptr** 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.

For example, the following code sequence is valid with our allocator, but not with the traditional malloc and free:

```
int *ptr;

// The returned memory object is between ptr and ptr+49
if ((ptr = (int *)Mem_Alloc(50 * sizeof(int))) == NULL) exit(1);

// Could replace 30 with any value from 0 to 49.
Mem_Free(ptr+30);
```

Thus, in our implementation, we have a more sophisticated data structure than the traditional malloc to track the regions of memory allocated by **Mem_Alloc**.

Program Specifications:-

We are required to implement following functions:-

We have provided the prototypes for these functions in the file **mem.h** and our implementation of memory allocator is in file **mem.cpp.**

1) int Mem_Init(int sizeOfRegion)

2) void *Mem_Alloc(int size)

3) int Mem_Free(void *ptr)

4) int Mem_IsValid(void *ptr)

5) int Mem_GetSize(void *ptr)

- ## Working of our Implementation:

We have used the following variables and structure to implement the malloc functions:-

a) struct node *allocatedlist_start

b) struct node *allocatedlist_end

c) struct node *freelist_start

d) struct node *freelist_end

e) int allocatednode_count; //**Initially 0**

f) int freenode_count; //**Initially 1**

g) struct node

{ void *dataptr;

int size; };

Our implementation is as follows:-

- We request 4 times the memory the user requests, we use one portion of it to store data and allocate the blocks that the user requests. For each block that has been allocated or deallocated, a new node 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. Also the variables allocListStart & allocListEnd are used to maintain the start and end of the allocated list, and freeListStart & freeListEnd are used to maintain the start and end of the free list.
  Following are the details of functions (with code snippets) that we implemented:-

1) 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().

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

This is how we implemented it in our code:

```
 if(sizeOfRegion<=0)
    {
    handle_error("Invalid size...!");
    return -1;
    }
//If size is less than 0, show error msg.
     sizegarbage=sizeOfRegion;
```

```
    int fd = open("/dev/zero", O_RDWR);
    ptr = mmap(NULL, 4*sizeOfRegion, PROT_READ |
PROT_WRITE, MAP_PRIVATE, fd, 0);      // getting memory
from OS
    if (ptr == MAP_FAILED)
    {
       handle_error("MMAP Error...!");
       return -1;
    }
    current_space=sizeOfRegion;
    freelist_start=(node*)(ptr + 4*sizeOfRegion -
sizeof(node));
    allocatedlist_start=(node*)(ptr + sizeOfRegion);
    allocatedlist_start->size=0;
    freelist_start->dataptr=ptr;
    freelist_start->size=sizeOfRegion;

    freelist_end=freelist_start;
    allocatedlist_end=allocatedlist_start;
    close(fd);
    return 1;
```

2) 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 "**FIRST FIT**" to find the free block in the memory and allocate it to the user. First we check the available present space and if the requested size is less than its value, then we move ahead. We find the first free available block in our region by traversing the free list.

- If the user requests more memory than the size of all the nodes available in the list, then,
  In this case, we display “Not Enough Memory” message and return a NULL pointer.

- Otherwise, if the space is available i.e. we find the node/block with size more than or equal to requested size, then, we create a node in the allocated list for that respective allocated block and initialize it with proper values. (Reduce the **free_node_count** by 1 and increment the **allocated_node_count**) .

This is how we implemented it in our code:

```
if(size>current_space)
    {
        handle_error("Not enough space
available...!\n");
        return NULL;
    }
```

//If size greater than current space then show error msg.

```
    node *firstlarge_node=freelist_start;
    node *currentnode=firstlarge_node;

    // pointer to be returned to user
    void *result_ptr=firstlarge_node->dataptr;

    int largenode_size=firstlarge_node->size;

    // first fit to search free node
    for(int i=1;i<=freenode_count;i++)
    {

        if(size < currentnode->size &&
firstlarge_node->dataptr > currentnode->dataptr)
            firstlarge_node=currentnode;
```

//Here we are using first fit policy

```
        currentnode--;
    }

    result_ptr=firstlarge_node->dataptr;
    node *newallocated_node;

    // if space is partially allocated from a free
node
    if(size<firstlarge_node->size)
    {

        firstlarge_node->size-=size;
        firstlarge_node->dataptr+=size;

        if(allocatedlist_start->size != 0)
            newallocated_node=allocatedlist_end +1;

        else
            newallocated_node=allocatedlist_start;

        newallocated_node->size=size;
        newallocated_node->dataptr=result_ptr;

        // increase num of allocated nodes
        allocatednode_count++;
        allocatedlist_end=newallocated_node;

        // update current space
        current_space-=size;
    }
    // if space is completely allocated of a free
node
    else if(size==firstlarge_node->size)
    {

        if(allocatedlist_start->size != 0)
```

```
            newallocated_node=allocatedlist_end +
1;
        else
            newallocated_node=allocatedlist_start;

        newallocated_node->size=size;
        newallocated_node->dataptr=result_ptr;
        allocatednode_count++;
        allocatedlist_end=newallocated_node;
        current_space-=size;

        if(firstlarge_node!=freelist_end)
        {
            firstlarge_node->dataptr=freelist_end-
>dataptr;
            firstlarge_node->size=freelist_end-
>size;
        }

        // increase num of free nodes
        freelist_end++;
        if(freenode_count>1)
            freenode_count--;
        else if(freenode_count==1)
        {
            freelist_start->dataptr=NULL;
            freelist_start->size=0;
        }


    }

    // if biggest node is less than size required
    else
    {
        handle_error("Not enough space
available...!\n");
        return NULL;
    }
    return result_ptr;
```

3) int Mem_Free(void *ptr) :

Mem_Free() frees the memory object that ptr1 falls within, according to the rules described in Project Objective.

- ➔ If ptr is NULL, then no operation is performed.
- ➔ The function returns 0 on success and -1 if ptr does not fall within the allocated memory.

In our implementation:-

When the value of ptr is NULL,
Then, we **return -1** as it isn't valid request for deallocation.

- ➔ We check the value of ptr with **Mem_isValid()** to check the validity of ptr.
- ➔ If Mem_isValid() returns false, then its not a a valid pointer to free and thus we display the error message and return -1.
- ➔ If ptr1 is valid by Mem_isValid(), then we traverse the whole list of allocated blocks and compare the lower bound and upper bound of the data pointer block to see if it lies in the range and find the node that contains this ptr.

- **When ptr specifies a partial range i.e. partially free the allocated block (not the whole block) :-**

We create a node for the free list and also reduce the size for the node of the allocated list.
And then we update **allocated_node_count** and **free_node_count** accordingly.

- **When ptr1 specifies the whole block to be free :-**

Here also we create a new node for the free list but we remove the node for the allocated block and update **allocated_node_count** and **free_node_count** accordingly.

- ➔ At the last of this function, we check that if their is any free region in either left or the right of current free region, then we merge that region to free region to make it bigger free region.

This is how we implemented it in our code:

```
if(ptr==NULL)
        return -1;

    //checking whether ptr provided by user is
within the valid range
    if(!Mem_IsValid(ptr))
    {
        handle_error("Invalid Pointer...!\n");

        return -1;
    }
    node *currentnode=allocatedlist_start;
    unsigned long long int low,high;

    //traversing the allocated list to find the
particular allocated node
    for(int i=0;i<allocatednode_count;i++)
    {

        low=(uintptr_t)currentnode->dataptr;
        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;

    node *newfree_node;
    //partially freeing a node
    if(currentnode->size!=1)
    {
        newfree_node=freelist_end;

        newfree_node--;
```

```
        newfree_node->size = actual_size -
currentnode->size;
    }

    //fully freeing a node
    else
    {
        newfree_node=freelist_end - 1;
        newfree_node->size = actual_size;
        currentnode->size=allocatedlist_end->size;
        currentnode->dataptr=allocatedlist_end-
>dataptr;
        allocatedlist_end--;
        allocatednode_count--;
    }
    newfree_node->dataptr=ptr1;
    freenode_count++;
    freelist_end=newfree_node;
    current_space+=newfree_node->size;
```

4) int Mem_isValid(void *ptr) :

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.

This is how we implemented it in our code:

```
node *currentnode=allocatedlist_start;
    unsigned long long int low,high;
    for(int i=0;i<allocatednode_count;i++)
    {
        low=(uintptr_t)currentnode->dataptr;
        high=low + currentnode->size;
```

```
        if((uintptr_t)low<=(uintptr_t)user_ptr &&
(uintptr_t)user_ptr<(uintptr_t)high)
            return 1;
        currentnode++;
    }
    return 0;
```

5) int Mem_GetSize(void *ptr) :

If **ptr** falls within the range of a currently allocated object, then this function returns the size in bytes of that object; otherwise, the function returns -1.

This is how we implemented it in our code:

```
node *currentnode=allocatedlist_start;
    unsigned long long int low,high;

    //traverse the allocated list to find the
particular allocated node
    for(int i=0;i<allocatednode_count;i++)

    {

        low=(uintptr_t)currentnode->dataptr;
        high=low + currentnode->size;

        if((uintptr_t)low<=(uintptr_t)ptr1 &&
(uintptr_t)ptr1 < (uintptr_t)high)
        {
```

```
            return currentnode->size;
        }
        currentnode++;
    }
    return -1;
```

- **End user documentation:-**

Our project contains the following files:

a) mem.h :

This file can be used directly in any program as the library, containing all the function declarations. (Header File)

b) mem.cpp :

This file contains our main implementation of **Mem_Alloc** and **Mem_free** i.e. definitions of all the functions. (Main code file)

c) main.cpp :

This is a kind of testing file used to check working of our implemented functions, which contains the main() function and using the “mem.h” as library(header file).

d) run.sh :

This script file is used to run the code and contains the lines for compiling and running the above made cpp files.

➔ Just type **./run.sh** in the current directory in the terminal to run the implemented code.