A1: Umalloc!

Himanshu Rathi (hr393) Shashwenth Muralidharan (sm2785)

1 Introduction

This document is a report on our implementation of a memory allocator that mimics the working of the malloc memory allocation function. The malloc() function allocates the requested memory and the free() function is used to deallocate the memory that was previously allocated by a memory allocator function. In this project, we implemented a umalloc() and ufree() function that replicates the working of malloc() and free() correspondingly. The memory in our project will be represented as a char array of size 10MB. Inside the memory each block contains its metadata before the block itself. So, the memory is represented as a series of address space containing metadata and . We will allocate and deallocate memory from this array.

In this report we will extensively detail the methods and API implemented, the data structures used and the performance metrics of our memory allocation library.

2 Methods

This section will provide all the necessary information regarding the functions we implemented in this library.

For simplicity, we have split this section into the following subsections based on their usage.

2.1 For Thread Execution

This section elucidates all the functions implemented for the purpose of memory allocation and deallocation, this includes:

1. umalloc()
2. ufree()

2.1.1 umalloc()

This function is used to allocate the requested memory in the array. This function takes the requested size as a parameter and returns the pointer to the memory address space back to the user. Initially, during the first call to the umalloc() the metadata will be initialized and the block will be allocated. Subsequently, all the umalloc() calls will be allocated based on a first-fit pattern.

One important thing to note is that the total memory allocated from the system's prespective is equal to the sum of the requested memory size and metadata size. But, however from the user's perspective it's just the size of the block. The memory is checked sequentially for an unallocated block, and if a suitable block is found the memory is allocated, and the pointer to the start of the memory is returned.

Furthermore, if the block of memory found after a sequential scan is greater than the requested memory, we will allocate only the necessary memory and assign the remaining memory as free i.e unallocated, in other words, split into another block with block size and that of the remaining memory and mark it as unallocated. While doing so, in our implementation, we have included various exception-handling mechanisms like:

1. Checks if the requested memory size exceeds the available memory.
2. To check if an invalid size request is made.

3. If there is no free memory available to be allocated.

4. If there is no large enough block available to allocate requested size

5. Although the memory is not full, there is not enough memory for the allocation

If the requested memory is unable to be allocated we return a NULL pointer. We, also check for some inappropriate conditions like allocation of memory greater than the size of memory or allocation of 0 bytes of memory. During these cases, a NULL pointer will be returned.

2.1.2 ufree()

The main purpose of this function is to deallocate the memory space requested by the user. This function takes the pointer used to reference as the input and deallocates it. Initially, we will check if the requested deallocation is a valid operation. If any of the following cases is true we will print an error statement.

1. Check if the pointer is NULL.

2. Check if the pointer references an address space outside the memory.

3. Check if the pointer points to a address that was not allocated by umalloc.

4. Check if the already free()ed pointer is being free()ed again.

Now that we have cleared any undesirable inputs, the ufree() function first traverses the memory sequentially to find the corresponding metadata address of the address space to be deallocated. In the metadata, the status of the block will be turned to unallocated. By this, we mean that the data will not be affected during the ufree() function. Instead, during the next umalloc() function the new data will overwrite the existing data residing in this "unallocated block".

After freeing the passed address, we traverse the entire memory block from the beginning and combine the consecutive free()ed/unallocated blocks into a one single block to avoid any un-necessary fragments/chunks in the memory. This would make sure we always have the largest free block available(between allocated blocks) for next allocation.

3 Data Structures

This section explains the important data structures involved in our implementation

3.1 MetaData Block

The meta-data is represented as a struct data structure with 2 elements. The first value, block_size, represents the requested block size. The second value, block_allocated, is used to represent if the block represented by the meta data is allocated or not.
For every allocation, the metadata is stored at the start of the block, after which the actual requested allocation happens.

```
typedef struct block_info
    {
    unsigned int block_size;
    char block_allocated;
    } block_info;
```

3.2 Memory Array

In our implementation, we will represent the memory as a char array. The MEM_SIZE in our implementation is 10MB.

```
static char mem[MEM_SIZE];
```

3.3 Address Array

This array is used in the memgrind.c program to store all the pointers of allocated blocks. This allows us to easily keep track of the allocated memory blocks in the memgrind.c program.

char *ptr[MAX];

4 Basic Integration

This section will provide all the necessary information regarding the functions we implemented in this library.

1. Consistency
2. Maximization
3. Basic Coalescence
4. Saturation
5. Time Overhead
6. Intermediate Coalescence

4.1 Consistency

To check for consistency in our implementation, we first initialize a small black of memory and check the address where it is allocated. We then free the block and repeat the same steps once again. From the output below, we can infer that our implementation is consistent because both allocations are allocated in the same memory location.

```
***********Testing Consistency************
Initalizing a
Starting address of a is 0x5587e889b088
Initalizing b
Starting address of b is 0x5587e889b088

Consitency is proven since both the blocks are assigned at the same Address

******************************************
```

Figure 1: Output of Consistency Implementation

4.2 Maximization

The idea here is to find the biggest memory size that can be allocated in a single umalloc() call.

4.2.1 Approach 1

In the first approach, we first double the valid value until it becomes invalid. Let the last valid value be "x". This x is returned as the maximum value of x.

```
***********Testing Maximization***********
Error in memgrind.c at line 74: Requested Memory Exceeds the available Memory

Maximum size is 8388608

****************************************
```

Figure 2: Approach 1: Maximization Output

4.2.2 Approach 2

In this approach we first double the valid value until it becomes invalid. Let the last valid value be "x". Then we allocate the first valid value we obtain after we keep halving the size x. The maximal value that can be allocated is "9437184" bytes in this approach. (The implementation of this approach is included as the last section in memgrind.c file).

```
***********Testing Maximization***********
Error in memgrind.c at line 33: Requested Memory Exceeds the available Memory
Error in memgrind.c at line 51: No Free memory available for allocation
Error in memgrind.c at line 51: No Free memory available for allocation

Maximum size is 9437184

****************************************
```

Figure 3: Approach 2: Maximization Output

4.3 Basic Coalescence

Here we will initially allocate half the maximum size followed by one-fourth of the maximum size. Then we will free both allocations and try to allocate the maximum size. The image below represents the basic coalescence operation and its corresponding outputs sequentially.

```
********Testing Basic Coalescence*********

Starting address of Half the size initialization is 0x5587e889b088
Starting address of Quarter the size initialization is 0x5587e8d1b090
Starting address of Max size initialization is 0x5587e889b088

****************************************
```

Figure 4: Basic Coalescence Output

4.4 Saturation

The idea of saturation is to fill the whole memory with allocations. Initially, 1024 bytes are allocated 9216 times and from there 1B allocations are made until the memory is full. In the output, the error in the first line represents that the memory is full i.e the memory is saturated.

```
************Testing Saturation***********
Error in memgrind.c at line 105: No Free memory available for allocation
The memory saturation is complete

****************************************
```

Figure 5: Output of Memory Saturation

4.5 Time Overhead

Here, we will calculate the amount of time taken to allocate 1B of memory to the last valid address space in the memory. We have represented the time taken in nano seconds. As we could see, from the implementation the maximum overhead is 217367 ns.

```
************Testing Time Overhead**********

Running time: 217367 nano seconds

**********************************************
```

Figure 6: Maximum Time Overhead

4.6 Intermediate Coalescence

Here we will initially, deallocate all the allocated blocks from the memory and finally initialize the maximal value back into the memory. From the output we could conclude that the maximal block is allocated in the first position after the deallocation of all the previous memory.

```
*****Testing Intermediate Coalescence*****

Address of Maximal block allocation is 0x5587e889b088

******************************************
```

Figure 7: Intermediate Coalescence Output

5 Error Detection

Error detection is an integral part in any implementation. By using (LINE & FILE) in our implementation we have incorporated various error detection mechanisms. There are a few examples below showing the execution of our error detection process.

5.1 Allocation of Invalid Block Size

Here we try to allocate an invalid block size. First, we try to initialize a block of size 0 and then we try to allocate a block of size greater than memory.

```
*****Trying to allocate invalid size of blocks*****

Error in errorhandling.c at line 11: Size to be allocated should be greater than 0
Error in errorhandling.c at line 12: Requested Memory Exceeds the available Memory

*****************************************
```

5.2 Freeing Unallocated Address

This example shows the error that occurs when we try to free unallocated address space.

```
*****Freeing address that wasn't allocated by malloc()*****

Error in errorhandling.c at line 24: Attempted to free an incorrect memory location

*****************************************
```

5.3 Redundant Freeing of Same Pointer

This error is thrown when the user tries to free a pointer that was already freed.

```
*****Redundant Freeing of Same pointer*****

Error in errorhandling.c at line 32: Address Passed is already Free()ed

*****************************************
```

5.4 Unavailable Requested Size Block

This error happens when there is free memory available but it is not large enough to satisfy the user's request.

```
*****Free Mem available, but Not large enough for block*****

Error in errorhandling.c at line 38: No Free memory available for allocation

*****************************************
```

5.5 Requested Size More Than Available Memory

This error occurs when the requested size is greater than the amount of free memory available.

```
****Requested Size is more than the available Free memory test case****

Error in umalloc.c at line 246: Requested Size is more than the available Free memory

*************************************************************************
```

5.6 No Block Available

This error happens when there is no block in the memory available to allocate the requested size.

```
****No block available to allocate the requested Size test case****

Error in umalloc.c at line 262: No block available to allocate the requested Size

****************************************
```

5.7 No Free Memory Available

This error happens whenever there is no free memory available for the allocation of the requested memory space.

```
****No Free Memory Available Test case****

Error in umalloc.c at line 272: No Free memory available for allocation

****************************************
```

6 Conclusion

From the above results, we can infer that our implementation of umalloc() has successfully executed all the basic benchmarks and also has proven consistent throughout all the test cases. Furthermore, we have shown how error handling is carried out using the help of macros.