CS-518: Assignment 1

Umalloc!

Tanya Sharma (tds104) and Sanchay Kanade (sk2656)

1. INTRODUCTION

This project aims at creating a user-defined memory allocator that uses the C library function malloc() as a template. In this scenario, we view memory as an array of bytes and make custom design choices about the organization, updating, and manipulation of memory using our user-defined memory allocator. Consequently, we also create a user definition for the C library function free() to deallocate the memory that had been previously allocated using our version of malloc().

2. DESIGN

The capacity of our mem array is 10,485,760 bytes which can be addressed 4 bytes of meta data with room to spare. However, for some smaller allocations we observed we need fewer bits to address them. We will be storing our meta data in place but our **meta data size will be variable** so that we can optimize our space and allocate a greater number of blocks.

We are appending 3 additional bits at the end of our meta data. The last 3 bits will store data about the meta data size and allocation of block.

1. The LSB or the rightmost bit of meta data will signify if the current chunk is allocated or not. (0/1: allocated/not allocated)
2. The second last and third rightmost bits of meta data will tell us about the size of bytes needed by metadata. Return type of our function will also depend on these bits as we need such windows to read that many bytes of meta data.

Size of Meta Data(bytes)	Return type of our function	Second and third Last Bits
1 byte	char	00
2 bytes	short	01
4 bytes	int	11

With this meta data structure, we can have 1 byte of meta data for payloads less than 32 bytes, 2 bytes of meta data for payloads less than 8192 bytes and 4 bytes for the rest of the sizes requested.

The advantage of this structure of meta data is that we can efficiently manage the given memory space and allocate a greater number of blocks within it (524287 blocks as verified in the results of saturation, time overhead and intermediate coalescence functions), but the disadvantage is that our speed for these operations is reduced which increases our running time.

3. DATA STRUCTURES

- **static char mem[CAPACITY];**
  Character array representation of memory to which space will be allocated to and deallocated from using our memory allocation and deallocation functions. CAPACITY is the size of the memory in bytes which is equivalent to 10MB in our project.

- **char init;**
  Variable that passes information to our user-defined functions about the state of memory and whether it has been in use before or does it need to be allocated for the first time. This information is used in decided whether metadata blocks need to be initialized for memory allocation. If init is d it means memory is in default sate and has not yet been initialized and if init is "i" that means memory is previously initialized.

- **unsigned int sizeRequested;**
  This variable denotes the size of the block that needs to be allocated which is payload plus meta data bytes.

- **unsigned char* currentBlock;**
  This variable denotes the pointer to the block which is under consideration.

- **unsigned int sizeOfCurrentBlock;**
  It tells the size of the current block under consideration. This variable is used to check if the free block is sufficient for the current block or not.

- **char* prevFree:**
  Pointer to the previous block of memory that is used when we are calling the function free().

4. UMALLOC()

The Umalloc() function is a user-defined memory allocator function that is used instead of malloc() in our project. The function takes in size of the block to be allocated as a parameter and returns the total size of the allocated block including metadata. According to the size of memory that needs to be allocated, we dynamically decide the size of metadata needed for that block. For sizes less than 32 bytes, we only require 1 byte of metadata, for sizes between 32 and 8191 bytes, we require 2 bytes of metadata and for sizes larger than that we need only 4 bytes of metadata. If the current allocation is the first one, we start by pointing our currentBlock to the first address in our memory address space at mem[0]. As long as we have enough memory to allocate, we set the current memory address and also pre-emptively set the block after our block to 0 since it has not been allocated yet. If there isn't enough memory for allocation, we return NULL. If the current allocation is not the first one, we use the first fit policy to find the first block of memory that can be used for allocation. When we reach a block that has not been allocated, there are two possibilities: either we have traversed to the end and there is no memory left to be allocated or we are between chunks of used memory. If the latter is true, we determine the size of the chunk (space between two allocated blocks) to see if the length is sufficient for our allocation. If so, we go ahead with allocation, or else we return NULL if sufficient memory is not present.

umalloc() function can handle the following errors:

- Not enough free memory for the block requested.
- Enough free memory but cannot fit requested block.
- There is no free memory/overflow.

5. UFREE()

This function is the user defined memory deallocation function. When we want to remove a block from our memory, we call ufree() to deallocate that block so that it can be used by someone else. The function takes a pointer to the memory address to be deallocated as a parameter and does not return any value. If the pointer points to a block of memory that is not allocated, we do not allow it to be deallocated again and print an error message with details about the file and line in which this error was encountered. If the pointer points to the correct block of memory to be deallocated, after deallocation, we check the previous block of memory to which this block is contiguous, and if that block is also free, then we merge both blocks to create one large block. This is done, so that when a umalloc() is called for a bigger size of memory, it will be able to find one continuous block of memory to satisfy its first fit approach.

ufree() function can report and handle the following errors:

- Freeing address which are not allocated my umalloc or not pointers.
- Deallocating memory which is already deallocated.
- Attempting to deallocate an invalid pointer.

6. HELPER FUNCTIONS

- **void prettyPrint();**
  Helper function to print details associated with the current block that has been allocated. It displays information such as the index, address, allocation status, metadata size and total size of the block.

- **int isAllocated(unsigned char* currentBlock);**
  A function that takes in the address of the current block as a parameter and checks whether it is allocated or not. In our project, we are using the rightmost bit of metadata to indicate whether a block is allocated. If the value of the rightmost bit is 1, then this represents that it is already allocated and if the value of the rightmost bit is 0, then it represents that the block is free. By doing a simple logical & with the rightmost bit of metadata, we can find out the state of the block.

- **void createBlock(unsigned char* currentBlock, unsigned short allocated, unsigned int size);**
  This method sets the 3 rightmost bits of our metadata to values that determine the length of metadata needed to store the address of a particular location. We know from the function isAllocated() that the rightmost bit is used to store whether a block of memory is in use or not, the remaining two bits from the right indicate the length of metadata. 00 denotes metadata of only 1 byte and 01 and 11 mean 2 and 4 bytes each.

- **int metadataSize(unsigned char* currentBlock);**
  This function returns the size of metadata needed by a given block of memory, in bytes, by analysing its rightmost 3 bits.

- **int payloadSize(unsigned char* currentBlock);**
  Helper function that returns the size of the address stored in metadata.

7. ANALYSIS

We know analyze how each function in our memgrind.c file works for our project to determine the correctness and efficiency of our design for different memory allocation related scenarios.

- **Consistency**
  To demonstrate that the same operation carried out twice by two different agents will produce the same result, we first allocate and deallocate some memory and then try performing the same operation again using a different pointer.
  Status: Successful

```
tds104@cpp:~/Desktop/Assignment/518_A1_Umalloc/code$ ./mem
Allocating memory
Populating memory
Address is -1722109887
Freeing  up memory
Re-allocating memory
Re-Populating memory
Address is -1722109887
Consistancy test successful.......
```

- **Maximization (Simple Coalescence)**
  This function demonstrates allocation and deallocation of blocks of memory that are increasing in size till it reaches a maximum size and then decreasing the size.
  Status: Successful

```
tds104@cpp:~/Desktop/Assignment/518_A1_Umalloc/code$ ./mem
........Reached Maximization........ 8388608
Maximization test successful :)
```

- **Basic Coalescence**
  We first allocate one half of our total memory followed by one-quarter of the size of memory. After deallocating both blocks, we try to allocate our maximum size to check if deallocation has been successful.
  Status: Successful

```
tds104@cpp:~/Desktop/Assignment/518_A1_Umalloc/code$ ./mem
.........Starting Basic Coalescence test.........
Allocating half of maximal memory size
Address of first pointer:469188676
Allocating a quarterof maximal memory size
Address of second pointer:474431560
Freeing up first pointer
Freeing up second pointe
Allocating maximal memory
Address of third pointer:469188676
.........Basic Coalescence succcessful.........INDEX: 0, ADDRESS: 469188672
    is Allocated: 1, METADATA: 4, TOTAL_SIZE: 10485760
```

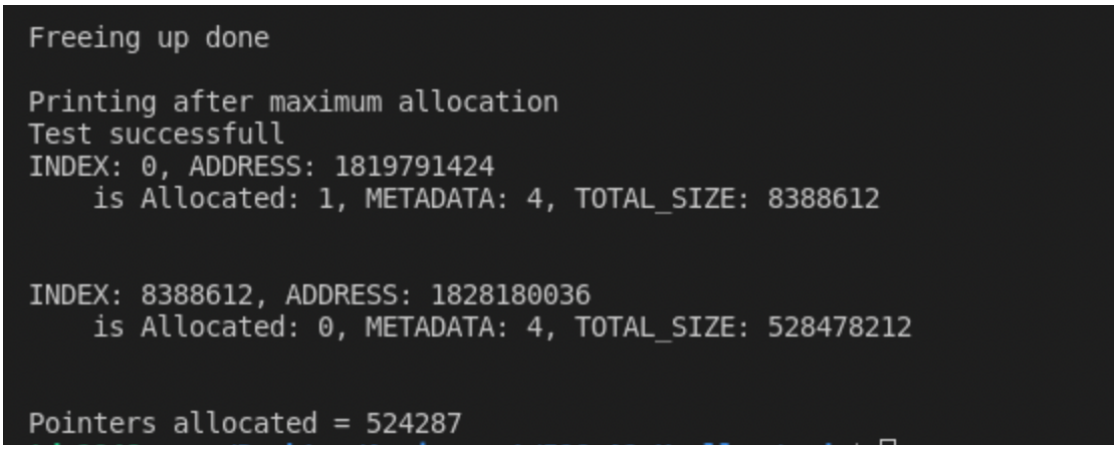
- **Saturation**
  After allocating 9K 1KB blocks into memory, we change our block size to 1B and keep allocating blocks of memory till our memory allocation function responds with a NULL when all the memory has been used up. This saturates our entire memory.
  Status: Successful

```
tds104@cpp:~/Desktop/Assignment/518_A1_Umalloc/code$ ./mem
......Starting saturation test.....

*************9K DONE*************


There is no free memory. Overflow in line 105,memgrind.c
.........Saturation test complete...........
```

- **Time Overhead**
  This method finds out the time it takes to allocate and deallocate 1B of memory at the end of our memory space. To do so, we saturate our memory and deallocate just the last 1B of memory to perform our operations on. It takes 2 micro seconds.
  Status: Successful

```
tds104@cpp:~/Desktop/Assignment/518_A1_Umalloc/code$ ./mem
..........Starting saturation test..........
.....Finished allocating 9k blocks of 1024.....
Time taken for 9k allocations is:470333
:There is no free memory. Overflow in line 141,memgrind.c
Time taken for saturation with 1B blocks is:-671892
..........Saturation test complete.........
ptr value is = 0
prev value is = 1452154943
..........Starting Time Overhead Test..........
..........Freeing up the last 1B block of memory..........

Max time overhead:2
 micro seconds
pointer count is = 524289
```

- **Intermediate Coalescence**
  After saturating all our memory, we attempt to free each allocation with blocks of size 1B. To check if deallocation was successful, we further attempt to allocate our maximum memory size.
  Status: Successful



As we can see from the results, the total number of pointers allocated or the total number of blocks allocated after saturation is 524287 which is due to variable meta data size but it takes approximately 19-21 mins to complete. Hence, due to increased number of allocations we are **prioritizing efficient memory space management over speed.**

Note:
Time Overhead and Intermediate coalescence function in our memgrind.c file already contain saturation for the ease of implementation and operation so no need to run saturation test individually. These individual functions can be run to complete respective test along with the saturation.