Assignment – 1 Design Document

Name: Kalahasti V V Deepesh

Roll Number: IMT2019508

Aim of the assignment:

Implement the following memory management functions for storage allocation on a heap by allocating an array on a global space. Write sample programs which test these functions rigorously.

Functions to be implemented:

a) void* my_calloc (size_t nmemb, size_t size):

   This function allocates space for an array of nmenb objects, each of whose size is ‘size’. The space is initialized to all bits zero.

b) void my_free (void* ptr):

   This function causes the space pointed to by ‘ptr’ to be deallocated.

c) void* my_malloc (size_t size):

   This function allocates space for an object whose size is specified by ‘size’ and whose value is indeterminate.

d) size_t my_heap_free_space(void):

   This function returns the total size of free space available in your own heap space from where you are allocating.

Information about Source Code and Execution of the program:

All the code for the assignment is written in three files main.c, my_functions.c and my_functions.h.

main.c contains the code for the ‘main’ function and all the test cases and the code for outputs is written here.

my_functions.h contains function declarations for all the functions that are need to complete the assignment.

my_functions.c contains all the code for the functions declared in the my_functions.h file.

Compile the programs together using gcc and execute to get the output.

Structure of the memory used:

Code snippets:

memory_size = 80000;

my_heap_ptr = (void*) malloc(memory_size);

start = (void*) my_heap_ptr;

Explanation:

memory_size is to specify the total number of bytes that will serve as our main memory array from which we will implement and test our malloc library implementation.

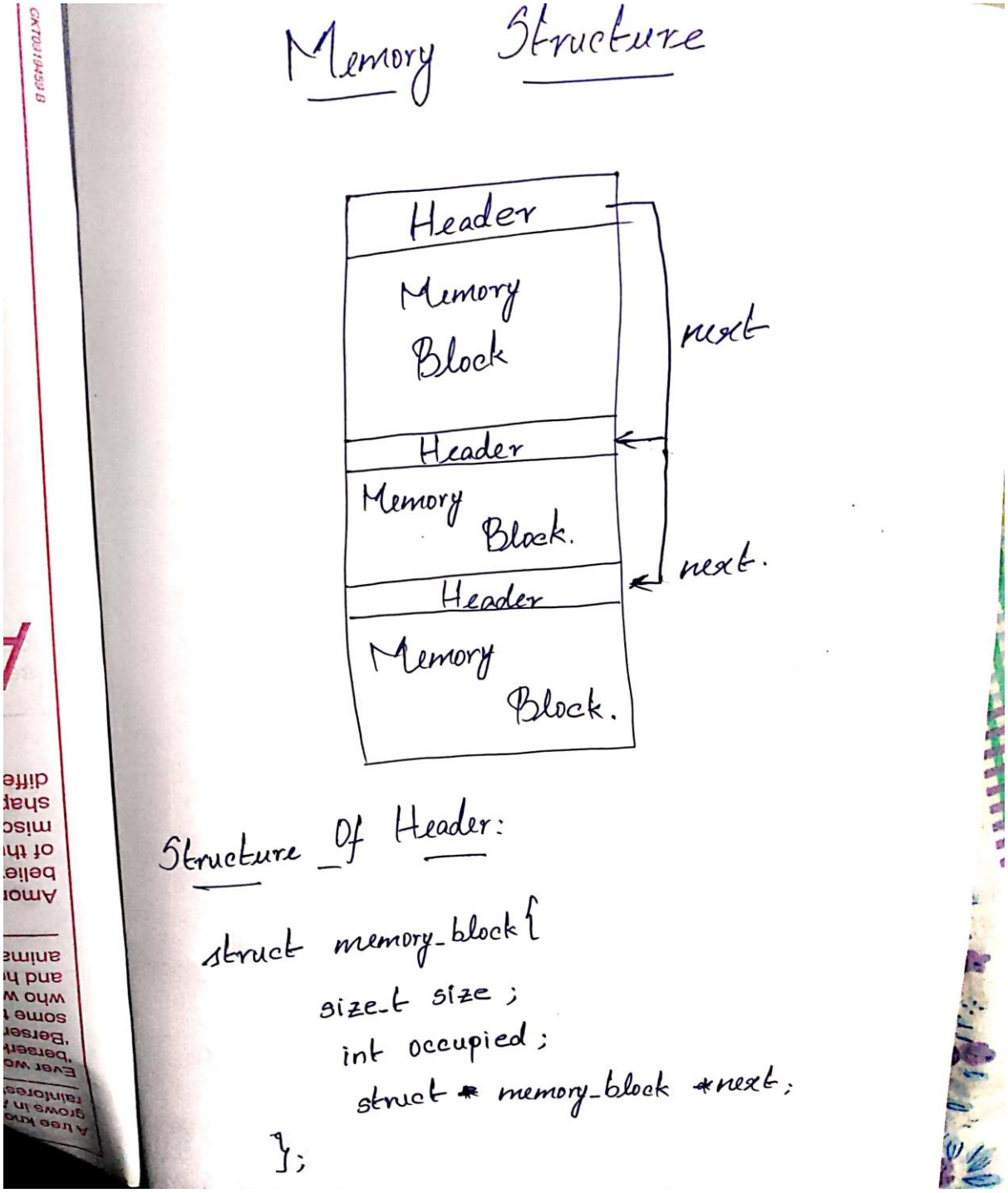
my_heap_ptr will be name of the array which will serve as out main memory. I have allocated memory for this array using the ‘malloc’ function in the C library.

start will be the header which will always be the header for the starting memory block which we will use to traverse our main memory.

Explanation about memory and header structure:

The information about the block of memory that has been allocated will be present in the header that resides right on top of the memory block that has been allocated.

The headers together form a linked list so that we can traverse through all the memory blocks that have been created.



Memory and header structure

Explanation about all of the functions:

1. initialise_memory ();

Initially when we first allocate the memory to be used by my functions it does not contain any header information. So, we need to call this function so that a header is placed on top of our memory structure which can then be used by other functions.

Code snippet:

```
void initialise_memory()
{
        start -> block_size = (memory_size) - sizeof(struct memory_block);
        start -> occupied = 0;
        start -> next = NULL;
        printf("Memory initialised\n");
}
```

Explanation:

This sets the state of the memory. The total size of the memory block is the size of the memory subtracted by the size of the header.

The memory block is not occupied hence its value is 0.

There is no other memory block so the next pointer points to NULL.

2. void split_memory(struct memory_block* required, size_t size);

If we encounter a memory block whose size is more than what is required then we need to split memory block.

Code snippet:

```
void split_memory (struct memory_block* required, size_t size)
{
```

```
	struct memory_block *new_block = (void*) ((void*) required + size +
sizeof (struct memory_block));

	new_block -> block_size = (required -> block_size) - size - sizeof (struct
memory_block);

	new_block -> occupied = 0;

	new_block -> next = required -> next;

	required -> block_size = size;

	required -> occupied = 1;

	required -> next = new_block;

}
```

Explanation:

This is achieved by placing a new header in front of the amount of space that we need so that the one large memory block is split into two blocks one with the required size.

3. void merge_memory ();

If two blocks are both unallocated then we need to merge into a single block of space so that space is conserved. We call this function after we free memory so that any consecutive free blocks that are created are merged together. This will conserve space and improve efficiency during future memory allocation calls.

Code snippet:

```
void merge_memory ()

{

	struct memory_block *previous_block, *current_block;

	current_block = start;

	while (current_block -> next != NULL)

	{

		if ((current_block -> occupied == 0) && (current_block -> next ->
occupied == 0))
```

```
                {
                        current_block -> block_size = current_block -> block_size +
current_block -> next -> block_size + sizeof (struct memory_block);
                        current_block -> next = current_block -> next -> next;
                }
                previous_block = current_block;
                current_block = current_block -> next;
        }
}
```

Explanation:

While looping through all of the header in the memory if we find any two consecutive blocks of memory which are both unoccupied then we merge them into one block by removing the second header and updating the size in the first header.

4. void* my_malloc (size_t size);

This is the implementation my malloc function. It works on the basis of “first fit algorithm”.

First fit algorithm:

We loop through all the memory blocks in the main memory.

We keep looping through the array until we find a memory block whose size in greater than or equal to the required size and which is un-occupied.

If we cannot find a memory block which satisfies the given conditions, we return a NULL pointer.

If the memory block we find has the size which is exactly equal to the required size, then we can return the pointer to the memory block without performing any extra operations.

If we find a memory block whose size is more than the required size then we split the memory block using the split_memory function and then return the pointer to the memory block.

Code snippet:

```
void* my_malloc (size_t size)
{
	void *return_pointer;
	struct memory_block *previous_block, *current_block;
	current_block = start;
	while((current_block -> block_size < size || current_block -> occupied == 1) && (current_block -> next != NULL))
	{
		previous_block = current_block;
		current_block = current_block -> next;
		printf("Passed over a undesirable block of memory\n");
	}
	if(current_block -> occupied == 0 && current_block -> block_size == size)
	{
		current_block -> occupied = 1;
		current_block++;
		return_pointer = (void*)current_block;
		printf("Block whose size is exactly what we need is found\n");
		return return_pointer;
	}
	else if(current_block -> occupied == 0 && current_block -> block_size > size + sizeof(struct memory_block))
	{
		split_memory(current_block, size);
		current_block++;
		return_pointer = (void*)current_block;
```

```
            printf("Block whose size is more than what we need is found\n");
            return return_pointer;
        }
        else
        {
            printf("Could not find a suitable block\n");
            return NULL;
        }
}
```

5. void my_free (void* ptr);

This function is my implementation of the free function in C library.

Code snippet:

```
void my_free (void* ptr)
{
        if (ptr != NULL)
        {
            struct memory_block *current_block = ptr;
            current_block--;
            current_block -> occupied = 0;
            merge_memory ();
        }
}
```

Explanation:

To free the space pointer to the by the pointer we go to the header of the memory block and set the occupied flag to 0 signifying that the memory block is free to use. We then call the merge_memory() function to merge any consecutive memory blocks which are unoccupied.

6. void* my_calloc (size_t nmemb, size_t size);

This is my implementation of the calloc function.

Code snippet:

```
void* my_calloc (size_t nmemb, size_t size)
{
	void* pointer = my_malloc (nmemb * size);
	memset (pointer, '\0', nmemb * size);
	return pointer;
}
```

Explanation:

For implementing the calloc function we just call the my_malloc function and use memset to set each of the bits in the memory block to be a zero bit and then return the pointer to the memory block.

7. size_t my_heap_free_space(void)

This function is used to get the total amount of free space in the main memory.

Code snippet:

```
size_t my_heap_free_space(void)
{
	size_t result = 0;
	struct memory_block *current_block;
	current_block = start;
	while (current_block != NULL)
	{
		if (current_block -> occupied == 0)
		{
			result += current_block -> block_size;
```

```
            }
            current_block = current_block -> next;
        }
        return result;
}
```

Explanation:

To get the total amount of free space in the memory we just loop through all of the memory headers.

If the header has the occupied flag as 0 that means that the memory block is free. So, we add the size of the memory that the header points to our final result.

Once we have looped through the whole linked list and added up all of the free space, we can return the result.

Test Cases used for Testing:

Test case 1:

We use my_malloc to allocate 40 bytes of memory and we type cast it to be a integer-pointer type. So, we should be able to put 10 integers in the memory block that we have just allocated.

This is demonstrated in the next part of the code where we put some integers in the memory block and then print them again to ensure they have been allocated properly.

We see a printed saying that we have encountered a block of memory larger than necessary. This is the starting memory header that we have initialised.

Test case 2:

We use my_malloc to allocate 10 bytes of memory and we type cast it to be a char-pointer type. So, we should be able to put 10 characters in the memory block that we have just allocated.

This is demonstrated in the next part of the code where we put some characters in the memory block and then print them again to ensure they have been allocated properly.

We see two lines printed in the console.

The first line says that we have encountered an undesirable block of memory. This is the integer memory block that we had allocated in the first test case.

The second line says we have found a memory with space larger than necessary. This is the space where we split the memory and allocate the memory block.

Test Case 3:

We use my_calloc to allocate 40 bytes of memory and we type cast it to be a integer-pointer type. So, now all the bytes have been initialised to have a 0 value.

We verify that all the values in assigned memory have a 0 value by printing them in the next part of the code.

The first two lines say that we have encountered an undesirable block of memory. This is the integer memory block and the character memory block that we have allocated in the first two test case.

Test Case 4:

We free the initial pointer that we had used in Test Case 1 using the my_free function.

Now we allocate 10 bytes of memory like in Test Case 2.

We see only one line printed now that states that we have found a memory block whose size is more than what is required. This is the memory block that we had initially allocated in the first test case that now been freed. So we are able to use that memory block now and are able to allocate that block.

Test Case Output: