

**Habib University**  
**Operating Systems - CS232**

**Assignment 03 - Report**  
**Stack and Heap Memory Management**



**Instructor:** Munzir Zafar

Ali Muhammad Asad - aa07190

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Memory Layout and Design</b>	<b>3</b>
2.1	Stack Memory Layout . . . . .	3
2.2	Heap Memory Layout . . . . .	4
<b>3</b>	<b>Makefile and I/O</b>	<b>4</b>
3.1	Makefile . . . . .	4
3.2	Input . . . . .	4
<b>4</b>	<b>Implementation</b>	<b>5</b>
4.1	Data Structures . . . . .	5
4.2	Algorithms and Code . . . . .	7
4.2.1	Main . . . . .	8
4.2.2	CF - Create Frame . . . . .	9
4.2.3	DF - Delete Frame . . . . .	10
4.2.4	Create <i>Type</i> Local Variables . . . . .	11
4.2.5	CH - Create Character Buffer on Heap . . . . .	12
4.2.6	DH - Deallocate a Heap Buffer . . . . .	13
4.2.7	SM - Show memory Image . . . . .	14
4.3	Assumptions and Stuff . . . . .	16
<b>5</b>	<b>Takeaway and Reflection</b>	<b>17</b>
<b>6</b>	<b>References</b>	<b>18</b>
<b>A</b>	<b>Appendix</b>	<b>19</b>

## 1 Introduction

The assignment simulates a simple stack and heap memory system.

1. The whole memory allocated is 500 bytes in size
2. The stack starts at the top of the memory, that is, at location 500, and grows towards the lower addresses.
3. The heap starts at the bottom of the memory, that is, at location 0 and grows towards higher addresses.
4. The stack can have a maximum of 5 frames; one frame per function.
5. The stack frame stores local variables, function return addresses and pointers.
6. The frame pointer points to the start address of the frame currently in execution.
7. The stack can grow up to a maximum of 200 bytes; downwards to a maximum of 300 bytes location.
8. The heap can grow up to a maximum of 300 bytes.

## 2 Memory Layout and Design

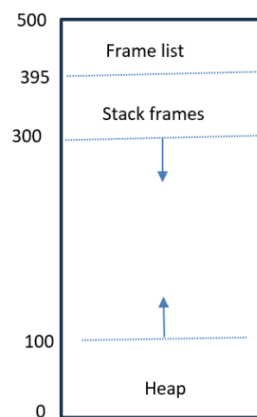


Figure 1: Memory Layout

The figure above depicts the functionality of the memory management system, and how the stack and heap may be visualized.

### 2.1 Stack Memory Layout

The stack is further divided into stack frames, and frame status. The frame status is 21 bytes (each) in size and contains metadata about its respective stack frame. The frame status stores the frame number (4 bytes), function name (8 bytes), function addresses (4 bytes), frame address (4 bytes), and a boolean value (1 byte) indicating whether that particular frame is in use or not.

Just after the frame status list in the stack, we have their respective stack frames (maximum of 5 frames). Each frame can store integers (4 bytes), doubles (8 bytes), character (1 byte each), and data pointers (4 bytes each). The minimum size of the stack frame is 10 bytes, and the maximum size of the stack frame is 80 bytes.

## 2.2 Heap Memory Layout

The heap can grow up to a maximum of 300 bytes. For each allocation, the system allocates the requested number of bytes plus 8 bytes (for storing the size of the allocated region and random generated magic number).

The system (not the memory but the system apart from the memory) maintains a free list which shows the memory segments that are currently free and not allocated. Along with this, the system also maintains an allocated list which shows the memory segments that are currently allocated and are not free. The free list and the allocated list are both implemented as linked lists.

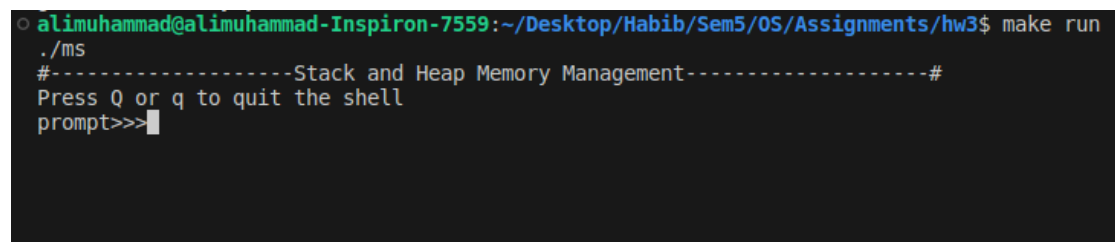
## 3 Makefile and I/O

### 3.1 Makefile

The accompanying makefile has also been provided with the program and supports the following commands:

- `make build`: builds the program and generates the executable file titled ‘`ms`’
- `make run`: runs the program on the command line where the inputs can be given
- `make clean`: cleans the directory by removing the executable file

### 3.2 Input

A terminal window with a dark background. The prompt is 'alimuhammad@alimuhammad-Inspiron-7559:~/Desktop/Habib/Sem5/OS/Assignments/hw3\$'. The user enters 'make run'. The prompt changes to './ms'. The program outputs a separator line '#-----Stack and Heap Memory Management-----#', followed by 'Press Q or q to quit the shell', and then a 'prompt>>>' with a cursor.

```
alimuhammad@alimuhammad-Inspiron-7559:~/Desktop/Habib/Sem5/OS/Assignments/hw3$ make run
./ms
#-----Stack and Heap Memory Management-----#
Press Q or q to quit the shell
prompt>>>|
```

Figure 2: Command Line Input

The above figure depicts the interactive shell that is opened when the executable is run. The user can input the commands (in correct format) to interact with the memory management system.

## 4 Implementation

### 4.1 Data Structures

#### 1. Stack Frame Metadata structure

```
1 struct __attribute__((__packed__)) framestatus{
2     int number; // frame number
3     char name[8]; // function name representing the frame
4     int functionaddress; // address of function in code section (will be
5     randomly generated in this case)
6     int frameaddress; // starting address of frame belonging to this
7     header in Stack
8     uint8_t used;
```

Listing 1: Stack Frame Metadata Structure

The above struct represents the previously talked about frame status structure. It is 21 bytes in size and contains metadata about its respective stack frame. The frame status stores the frame number (4 bytes), function name (8 bytes), function addresses (4 bytes), frame address (4 bytes), and a boolean value (1 byte) indicating whether that particular frame is in use or not. [I used a “uint8\_t” along with padding options for the compiler so that `used` only takes 1 byte, else it was taking more bytes due to the compiler adding padding.]

#### 2. FreeList and Allocated Structure

```
1 struct freelist{
2     int start; // start address of free region
3     int size; // size of free region
4     struct freelist* next; // pointer to next free region
5 };
6
7 struct allocated{
8     char name[8]; // name of the buffer on heap
9     int startaddress; // start address of buffer on heap
10    struct allocated* next; // pointer to next allocated buffer
11 };
```

Listing 2: FreeList and Allocated Structure

The above structs represent a single node of a free list structure - a dynamic linked list - and a single node of an allocated list structure - dynamic linked list. The free list structure is used to keep track of the free regions in the heap, and the allocated list structure keeps track of the allocated regions in the memory. Each time a new buffer is created in the heap, the free regions are traversed to check if there is a chunk large enough to create a buffer of the given size. If not found, an error is thrown, else the size and start address of the free region is updated and reduced accordingly. Similarly, a new node is created in the allocated regions list to keep track of the allocated region. When an allocated region is de-allocated, the node is removed from the allocated list and a new node is created in the free list.

### 3. Memory Structure

```
1 char memory[MEMSIZE];  
2 struct framestatus frameStatusList[MAX_FRAMES]; // frame status list
```

Listing 3: Memory Array

The first line above shows the memory array of size ‘MEMSIZE’ (defined in our code to be 500 bytes). So we simulate our memory using an array of type `char`, since a `char` is of 1 byte, therefore, 500 bytes memory is simulated. Each time data has to be written on the memory, respective indices are used to store data.

The second line shows an array ‘‘`frameStatusList`’’ of struct `framestatus` type. This array is used to store the frame status of each frame.

## 4.2 Algorithms and Code

For the implementation of the memory system, a character memory array was used along with a dynamic free list, allocated list, a frame counter, and a stack pointer. To remove the unnecessary passing of arguments and necessary variables each time a function had to be called, I decided to make those global, so that the whole program could access them. The below listing shows the global variables and the initialization function to initialize the structures:

```
1 char memory[MEMSIZE];
2 struct framestatus frameStatusList[MAX_FRAMES]; // frame status list
3 int frame_counter, top_StackFrame; // frame counter, stack pointer
4 struct freelist fl_node;
5 struct freelist* fl_head = &fl_node; // free list head
6 struct allocated a_node;
7 struct allocated* a_head = &a_node; // allocated list head
8
9 // Initialization Function
10 void init(){
11     srand(time(NULL)); // seed random number generator
12     frame_counter = 0; // initialize frame counter
13     top_StackFrame = 393; // initialize stack frame head
14
15     /* Initialize the Free List Head */
16     fl_head->start = 0;
17     fl_head->size = MAX_HEAP_SIZE;
18     fl_head->next = NULL;
19
20     /* Initialize the Allocated List */
21     strcpy(a_head->name, "");
22     a_head->startaddress = 0;
23     a_head->next = NULL;
24
25     /* Copy Contents to memory to complete memory initialization */
26     memcpy(&memory[394], &frameStatusList, sizeof(frameStatusList));
27 }
```

Listing 4: Global Variables and Initialization

The `frameStatusList` is made an array of size “MAX\_FRAMES” that was defined in the code to be 5. A framecounter was initialized to keep track of the number of frames, and a `top_StackFrame` to keep track of the address in memory to which the stack writes data on, thus acting as a pointer to the appropriate position in the stack; initialized to 393 since from index 394 to 499, we have the frame status list, so directly below this the stack starts growing downwards. The `fl_head` and `a_head` are the heads of the free list and allocated list respectively. The `memcpy()` function is used to copy the frame status list to the memory.

### 4.2.1 Main

The main function is the entry point of the program. It first calls the `init()` function to initialize the memory and data structures. Then it opens an interactive shell where the user can input the commands to interact with the memory system. The main function is shown below:

```

1 int main(){
2     printf("#-----Stack and Heap Memory Management
3     -----#\n");
4     printf("Press Q or q to quit the shell\n"); init();
5     while(true){
6         char input[3], name[8];
7         int functionaddress, intval, buffer_size;
8         double doubleval;
9         char charval;
10        printf("prompt>>>");
11        scanf("%s", input);
12        if(strcmp(input, "CF") == 0){
13            scanf("%s %d", name, &functionaddress);
14            CF(name, functionaddress);
15        }
16        else if(strcmp(input, "DF") == 0) DF();
17        else if(strcmp(input, "CI") == 0){
18            scanf("%s %d", name, &intval);
19            CI(name, intval);
20        }
21        else if(strcmp(input, "CD") == 0){
22            scanf("%s %lf", name, &doubleval);
23            CD(name, doubleval);
24        }
25        else if(strcmp(input, "CC") == 0){
26            scanf("%s %c", name, &charval);
27            CC(name, charval);
28        }
29        else if(strcmp(input, "CH") == 0){
30            scanf("%s %d", name, &buffer_size);
31            CH(name, buffer_size);
32        }
33        else if(strcmp(input, "DH") == 0){
34            scanf("%s", name);
35            DH(name);
36        }
37        else if(strcmp(input, "SM") == 0) SM();
38        else if(strcmp(input, "Q") == 0 || strcmp(input, "q") == 0) exit(
39            EXIT_SUCCESS);
40        else printf("Invalid input, please try again\n");
41    }
42    return 0;
43 }
```

Listing 5: Main Function

However, it has been assumed that the inputs will always be in the correct format and invalid commands will not be given to the shell.



### 4.2.2 CF - Create Frame

syntax: CF <function name> <function address>

```

1 void CF(char* functionname, int functionaddress){
2     if(strlen(functionname) > 8){
3         fprintf(stderr, "Error: Function name exceeds 8 characters\n");
4         return;
5     }
6     if(MEMSIZE - (top_StackFrame + 1) + MIN_FRAME_SIZE > MAX_STACK_SIZE){
7         fprintf(stderr, "Error: Stack Overflow, not enough memory
8         available for new function\n"); return;
9     }
10    if(frame_counter == MAX_FRAMES){
11        fprintf(stderr, "Error: Cannot create another frame, maximum
12        number of frames have been reached\n"); return;
13    }
14    for(int i = 0; i < frame_counter; i++){
15        if(strcmp(frameStatusList[i].name, functionname) == 0){
16            fprintf(stderr, "Error: Function already exists with the
17            given name\n"); return;
18        }
19    }
20    frameStatusList[frame_counter].used = true;
21    frameStatusList[frame_counter].number = frame_counter;
22    strncpy(frameStatusList[frame_counter].name, functionname, sizeof(
23    frameStatusList[frame_counter].name));
24    if(frame_counter > 0 && ((frameStatusList[frame_counter - 1].
25    frameaddress - top_StackFrame) < 10)) top_StackFrame = frameStatusList
26    [frame_counter - 1].frameaddress - 10;
27    frameStatusList[frame_counter].functionaddress = functionaddress;
28    frameStatusList[frame_counter].frameaddress = top_StackFrame;
29    memcpy(&memory[394], &frameStatusList, sizeof(frameStatusList));
30    frame_counter++;
31    printf("Created frame %s with address %d\n", functionname,
32    functionaddress); return;
33 }

```

Listing 6: Create Frame

The above function effectively creates a frame in the stack, and sets its metadata in the frame status list at the appropriate position. It checks if a frame can be created by checking if the stack is full or not, then checks if 5 frames have already been created, then checks if a frame with the given name already exists or not. If any check is failed, an error is thrown, else the frame is created by setting the appropriate metadata in the frame status list, and incrementing the frame counter. The `memcpy()` function is used to copy the frame status list to the memory, and a message is printed that the frame was created successfully.

### 4.2.3 DF - Delete Frame

syntax: DF

```
1 void DF(){
2     if(frame_counter == 0){
3         fprintf(stderr, "Error: Stack is empty, no frames to delete\n");
4         return;
5     }
6     for(int i = top_StackFrame; i < frameStatusList[frame_counter - 1].
7         frameaddress; i++) memory[i] = 0;
8     if(frame_counter == 1) top_StackFrame = 393;
9     else top_StackFrame = frameStatusList[frame_counter - 1].frameaddress;
10    frame_counter--;
11    frameStatusList[frame_counter] = (struct framestatus){0};
12    memcpy(&memory[394], &frameStatusList, sizeof(frameStatusList));
13    printf("Deleted Frame\n"); return;
14 }
```

Listing 7: Delete Frame

The DF works as follows; it first checks if there are any frames to delete or not. If no, an error is thrown, else it clears the memory from the stack pointer up till the start of the stack frame, and also updates the stack pointer accordingly. Then it decrements the frame counter, and sets the frame status list at the appropriate position to the default struct. The `memcpy()` function is used to copy the frame status list to the memory, and a message is printed that the frame was deleted successfully.

#### 4.2.4 Create *Type* Local Variables

syntax: CI <variable name> <value> for integers  
CD <variable name> <value> for doubles  
CC <variable name> <value> for characters

```
1 void CI(char* integername, int integervalue){
2     if(var_errors(1) == -1) return;
3     top_StackFrame -= sizeof(int);
4     memcpy(&memory[top_StackFrame], &integervalue, sizeof(int));
5     printf("Created integer %s with value %d\n", integername, integervalue)
6     ; return;
7 }
8 void CD(char* doublename, double doublevalue){
9     if(var_errors(2) == -1) return;
10    top_StackFrame -= sizeof(double);
11    memcpy(&memory[top_StackFrame], &doublevalue, sizeof(double));
12    printf("Created double %s with value %lf\n", doublename, doublevalue);
13    return;
14 }
15 void CC(char* charname, char charvalue){
16     if(var_errors(3) == -1) return;
17     top_StackFrame -= sizeof(char);
18     memcpy(&memory[top_StackFrame], &charvalue, sizeof(char));
19     printf("Created char %s with value %c\n", charname, charvalue); return;
20 }
```

Listing 8: Create Local Variables

The `var_errors(<var type>)` is a function designed to handle appropriate errors related to creation of local variables that will be attached within the appendix. Based on the type of variable, it checks if there are frames or not before creation of the variable, if there is enough space on the stack to accomodate a variable and if the current stack frame has enough space to accomodate a variable. If any check fails, an error is thrown, else the variable is created by decrementing the stack pointer by the size of the variable, and copying the value of the variable to the memory. A message is printed that the variable was created successfully.

#### 4.2.5 CH - Create Character Buffer on Heap

syntax: CH <buffer name> <buffer size>

```

1 void CH(char* buffername, int size){
2     if(size <= 0){
3         fprintf(stderr, "Error: Size of buffer cannot be less than or
4         equal to 0\n"); return;
5     }
6     int total_size = size + BUFFER_METADATA_SIZE;
7
8     if(frame_counter == 0){
9         fprintf(stderr, "Error: Stack is empty, no frames to create
10        pointer to buffer in\n"); return;
11    }
12    if(top_StackFrame - sizeof(int) < 300){
13        fprintf(stderr, "Error: Stack limit reached, not enough memory to
14        create a pointer to buffer\n"); return;
15    }
16    if(frameStatusList[frame_counter - 1].frameaddress - (top_StackFrame
17    - sizeof(int)) > 80){
18        fprintf(stderr, "Error: Frame is full, cannot create more data on
19        it\n"); return;
20    }
21
22    struct freelist* curr = fl_head;
23    while(curr != NULL && curr->size < total_size) curr = curr->next;
24    if(curr == NULL){
25        fprintf(stderr, "Error: The heap is full, cannot create more data
26        on it\n"); return;
27    }
28    int magic = rand(), heapStart = curr->start;
29    memcpy(&memory[heapStart], &size, sizeof(int));
30    memcpy(&memory[heapStart + sizeof(int)], &magic, sizeof(int));
31    for(int i = heapStart + BUFFER_METADATA_SIZE; i < heapStart +
32    total_size; i++) memory[i] = 'a' + (rand() % 26);
33
34    curr->start += total_size; curr->size -= total_size;
35
36    struct allocated* newAlloc = (struct allocated*) malloc(sizeof(struct
37    allocated));
38    strcpy(newAlloc->name, buffername);
39    newAlloc->startaddress = heapStart;
40    newAlloc->next = a_head->next;
41    a_head->next = newAlloc;
42
43    top_StackFrame -= sizeof(int);
44    int bufferAddress = heapStart;
45    memcpy(&memory[top_StackFrame], &bufferAddress, sizeof(int));
46
47    printf("Created buffer %s with size %d\n", buffername, size); return;
48 }

```

Listing 9: Create Character Buffer on Heap

This function creates a character buffer on the heap. It first checks if the size of the buffer is greater than 0 or not, then checks if there are frames or not, then checks if there is enough space on the stack to create a pointer to the buffer, then checks if the current stack frame has enough space to accomodate a pointer to the buffer, then checks if there is enough space on the heap to create a buffer of the given size. If any check fails, an error is thrown, else the buffer is created by finding a free region on the heap that is large enough to accomodate the buffer, and then creating the buffer by copying the size of the buffer and a random magic number to the start of the buffer, and then filling the buffer with random characters. The free region is then updated accordingly, and a new node is created in the allocated list to keep track of the allocated region. The stack pointer is then decremented by the size of an integer, and the address of the buffer is copied to the memory. A message is printed that the buffer was created successfully.

#### 4.2.6 DH - Deallocate a Heap Buffer

syntax: DH <buffer name>

```

1 void DH(char* buffername){
2     if(a_head == NULL){
3         fprintf(stderr, "Error: The pointer is NULL or already de-
4 allocated\n"); return;
5     }
6     struct allocated* prev = NULL, *curr = a_head;
7     while(curr != NULL && strcmp(curr->name, buffername) != 0){
8         prev = curr; curr = curr->next;
9     }
10    if(curr == NULL){
11        fprintf(stderr, "Error: The pointer is NULL or already de-
12 allocated\n"); return;
13    }
14    int bufferSize = *(int*)(memory + curr->startaddress);
15    memset(memory + curr->startaddress, 0, bufferSize +
16 BUFFER_METADATA_SIZE);
17
18    if(prev != NULL) prev->next = curr->next;
19    else a_head = curr->next
20    struct freelist* newFree = (struct freelist*) malloc(sizeof(struct
21 freelist));
22    newFree->start = curr->startaddress;
23    newFree->size = bufferSize + BUFFER_METADATA_SIZE;
24    newFree->next = fl_head; fl_head = newFree; free(curr);
25    printf("De-allocated buffer %s\n", buffername); return;
26 }

```

Listing 10: Deallocate a Heap Buffer

The DH function deallocates a heap buffer. It first checks if the allocated list is empty or not, then traverses the allocated list to check if the buffer with the given name exists or not. If not, an error is thrown, else the buffer is deallocated by setting the buffer to 0, and then creating a new node in the free list to keep track of the free region. The allocated node is then deleted and the allocated list updated accordingly. A message is printed that the buffer was deallocated successfully.

*\*Note: It was assumed that coalescing of contiguous memory was not a requirement.*

#### 4.2.7 SM - Show memory Image

syntax: SM

```

1 void SM(){
2     printf("#-----#\n");
3     for(int i = MEMSIZE; i >= 0; i--)
4         printf("%d: %X \n",i, memory[i]);
5
6     printf("\n");
7     printFreeList(); printAllocatedList();
8
9     printf("#-----#\n");
10 }
```

Listing 11: Show Memory

This function shows the snapshot of the memory at the current time from top to bottom. The memory image is displayed in hex values for each byte. It then prints the free list and allocated list. The `printFreeList()` and `printAllocatedList()` functions are defined in the appendix.

A few samples of the memory image can be seen below:

```

422: 0
421: 0
420: 0
419: 0
418: 0
417: 0
416: 0
415: 0
414: 1
413: 0
412: 0
411: 1
410: FFFFFFF89
409: 0
408: 0
407: 4
406: FFFFFFFD2
405: 0
404: 0
403: 0
402: 0
401: 6E
400: 69
399: 61
398: 6D
397: 0
396: 0
395: 0
394: 0
393: 0
392: 0
391: 0
390: 0
389: 7
388: 0
387: 0
386: 0
385: F
384: 61
383: 0
382: 0
381: 0
380: 0
379: 0
```

Figure 3: Snapshot Example 1

For the above example, the following commands were executed (in order of execution);

1. CF main 1234

2. CI int 7
3. CI int 15
4. CC char a.

The memory image shows the stack frame metadata, the stack frame, and the local variables created in the stack frame. Here's a breakdown of how the memory snapshot reflects the commands executed:

1. The first command creates a stack frame with the name "main" and the function address 1234. Since this is the first frame, frame number was set to 0, as per indexing conventions, therefore, from bytes 394 - 397 there is only a 0 (number stored here). The name is stored in the following 8 bytes, `main` translates to 6D, 61, 9, 6E for m, a, i, n respectively in hex (bytes 398 - 405). then we have the function address of 4 bytes (406 - 409). Frame address from 410 - 413. And finally the boolean value at 414 (1 byte). In total this is 21 bytes.
2. Next we have an integer "7" stored in the stack frame at 389 and takes 4 bytes.
3. Next an integer "15" translated to "F" in hex at 385 and takes 4 bytes.
4. Finally we have a character "a" translated to "61" in hex that takes 1 byte at 384.

An example for the buffer is shown below:

```

29: 0
28: 0
27: 0
26: 0
25: 0
24: 0
23: 0
22: 0
21: 0
20: 0
19: 73
18: 72
17: 3
16: 5
15: FFFFFFF7
14: 3C
13: 0
12: 0
11: 0
10: 2
9: 61
8: 66
7: 6F
6: 5C
5: 70
4: 50
3: 0
2: 0
1: 0
0: 2
Free List:
Start: 20, Size: 10
Start: 30, Size: 270

Allocated List:
Name: subbuf, Start Address: 10
Name: buff, Start Address: 0

```

For the above example, the following commands were executed (in order of execution);

1. CH buff 2
2. CH subbuf 2

### 3. CH buf3 2, DH buf3

The memory image shows the heap buffer metadata, the heap buffer, free list and allocated lists. Here's a breakdown of how the memory snapshot reflects the commands executed:

1. The first command creates a heap buffer with the name "buff" and size 2. The size is stored in the first 4 bytes (0 - 3). The magic number is stored in the next 4 bytes (4 - 7). The buffer is filled with random characters from 8 - 9 (2 bytes). The allocated list shows a buffer named "buff" at start address 0
2. The second command creates a heap buffer with the name "subbuf" and size 2. The size is stored in the first 4 bytes (10 - 13). The magic number is stored in the next 4 bytes (14 - 17). The buffer is filled with random characters from 18 - 19 (2 bytes). The allocated list shows a buffer named "subbuf" at start address 10
3. The next command creates a buffer "buff3", however, the "DH buff" command deletes the buffer, therefore, the buffer "buff3" cannot be seen on byte 20. However, since we are not coalescing, the free list is an indicator that there was a buffer at start address 20, of size 10 (2 + 8), however, it has been deleted, so we have a free region.

## 4.3 Assumptions and Stuff

While implementing the memory management system, a few assumptions were made. These assumptions are listed below:

- The inputs will always be in the correct format, and the user will not input invalid commands.
- The names will never exceed 8 characters of length (as per the assignment pdf).
- When deleting a stack, the corresponding data on the heap (if there are pointers pointing to data in the heap) is not cleared, as this shows a potential memory leak since the data on the heap is not being used anymore even though its pointer has been deleted.
- When deleting a buffer on the heap, the corresponding data pointer on the stack is not deleted, as this shows a potential dangling pointer since the pointer is pointing to a memory location that has been deallocated.
- The free list on the heap doesn't coalesce contiguous free regions, which also gives rise to the assumption that the user will always input the correct size of the buffer to be created on the heap unless they want to test if a buffer cannot be created when there are significant small chunks.
- The SM command (snapshot of memory image) doesn't show names or anything, rather just the hex values stored at respective bytes of the memory. If no data exists, a 0 will be shown.
- Instead of a space as said in the pdf, the memory was left empty and not initialized with any value, as then the SM function would display 20; the hex value when a space is translated in hex. This could lead to readability issues, therefore, the memory was left empty and 0s would be shown instead.



## 5 Takeaway and Reflection

The homework, and its implementation was quite easy and fun. However, to understand the homework and its implementation took quite a lot of time and visits to the faculty offices. The assignment pdf was not clear, quite vague at times and contradictory to itself which only became clear as the implementation progressed. However, once things became clear, the implementation itself didn't take quite a lot of time. It was a good learning experience overall.



## 6 References

### References

- [1] Remzi H. Arpaci-Desseau and Andrea C. Arpaci-Desseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, LLC, 2015.
- [2] *kuleuven-deipenbeek* [Online]. Available: <https://kuleuven-diepenbeek.github.io/osc-course/ch8-stack/stackvsheap/>
- [3] *GitHub* [Online]. Available: <https://github.com/D0D0123/MemoryGuideC/>
- [4] *Illesha* [Online]. Available: <https://llesha.github.io/programming/heap-and-stack/>
- [5] *Illinois.edu* [Online]. Available: <https://courses.grainger.illinois.edu/cs225/fa2021/resources/stack-heap/>
- [6] OpenAI. *ChatGPT* [Online]. Available: <https://chat.openai.com/> mainly for understanding, error resolving, commentation
- [7] Microsoft. *Bing AI Chat Bot* [Online]. Available: <https://www.bing.com/search?q=Bing+AI&showconv=1&FORM=hpcodx> mainly for understanding, error resolving, commentation.
- [8] Trips to Sir Tariq Kamal and Sir Munzir's office for resolving queries

## A Appendix

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<stdbool.h>
4 #include<time.h>
5 #include<string.h>
6 #include<stdint.h>
7
8 #define MEMSIZE 500
9 #define MAX_STACK_SIZE 200
10 #define MAX_HEAP_SIZE 300
11 #define MAX_FRAMES 5
12 #define MIN_FRAME_SIZE 10
13 #define MAX_FRAME_SIZE 80
14 #define BUFFER_METADATA_SIZE 8
15
16 // Frame metadata structure
17 struct __attribute__((__packed__)) framestatus{
18     int number; // frame number
19     char name[8]; // function name representing the frame
20     int functionaddress; // address of function in code section (will be
21         randomly generated in this case)
22     int frameaddress; // starting address of frame belonging to this header
23         in Stack
24     uint8_t used; // boolean value to check if frame is used or not
25 };
26
27 // Free list structure
28 struct freelist{
29     int start; // start address of free region
30     int size; // size of free region
31     struct freelist* next; // pointer to next free region
32 };
33
34 // Structure to store regions currently allocated on heap - for
35     performing allocations and deallocations
36 struct allocated{
37     char name[8]; // name of the buffer on heap
38     int startaddress; // start address of buffer on heap
39     struct allocated* next; // pointer to next allocated buffer
40 };
41
42 /* Global Variables and Initialization */
43 char memory[MEMSIZE];
44 struct framestatus frameStatusList[MAX_FRAMES]; // frame status list
45 int frame_counter, top_StackFrame; // frame counter, stack pointer
46 struct freelist fl_node;
47 struct freelist* fl_head = &fl_node; // free list head
48 struct allocated a_node;
49 struct allocated* a_head = &a_node; // allocated list head
50
51 // Initialization Function
```

```
49 void init(){
50     srand(time(NULL)); // seed random number generator
51     frame_counter = 0; // initialize frame counter
52     top_StackFrame = 393; // initialize stack frame head
53
54     /* Initialize the Free List Head */
55     fl_head->start = 0;
56     fl_head->size = MAX_HEAP_SIZE;
57     fl_head->next = NULL;
58
59     /* Initialize the Allocated List */
60     strcpy(a_head->name, "");
61     a_head->startaddress = 0;
62     a_head->next = NULL;
63
64     /* Copy Contents to memory to complete memory initialization */
65     memcpy(&memory[394], &frameStatusList, sizeof(frameStatusList));
66 }
67
68 // Prototypes for functions
69 void CF(char* functionname, int functionaddress);
70 void DF();
71 void CI(char* integername, int integervalue);
72 void CD(char* doublename, double doublevalue);
73 void CC(char* charname, char charvalue);
74 void CH(char* buffername, int size);
75 void DH(char* buffername);
76 void SM();
77
78 int main(){
79     printf("#-----Stack and Heap Memory Management
80     -----#\n");
81     printf("Press Q or q to quit the shell\n"); init();
82     while(true){
83         char input[3], name[8];
84         int functionaddress, intval, buffer_size;
85         double doubleval;
86         char charval;
87         printf("prompt>>>");
88         scanf("%s", input);
89         if(strcmp(input, "CF") == 0){
90             scanf("%s %d", name, &functionaddress);
91             CF(name, functionaddress);
92         }
93         else if(strcmp(input, "DF") == 0) DF();
94         else if(strcmp(input, "CI") == 0){
95             scanf("%s %d", name, &intval);
96             CI(name, intval);
97         }
98         else if(strcmp(input, "CD") == 0){
99             scanf("%s %lf", name, &doubleval);
100             CD(name, doubleval);
101         }
102     }
```

```
101     else if(strcmp(input, "CC") == 0){
102         scanf("%s %c", name, &charval);
103         CC(name, charval);
104     }
105     else if(strcmp(input, "CH") == 0){
106         scanf("%s %d", name, &buffer_size);
107         CH(name, buffer_size);
108     }
109     else if(strcmp(input, "DH") == 0){
110         scanf("%s", name);
111         DH(name);
112     }
113     else if(strcmp(input, "SM") == 0) SM();
114     else if(strcmp(input, "Q") == 0 || strcmp(input, "q") == 0) exit(
EXIT_SUCCESS);
115     else printf("Invalid input, please try again\n");
116 }
117 return 0;
118 }
119
120 int var_errors(int var_type){
121     if(frame_counter == 0){
122         fprintf(stderr, "Error: Stack is empty, no frames to create variable
in\n"); return -1;
123     }
124     if(var_type == 1){
125         if(top_StackFrame - sizeof(int) < 300){
126             fprintf(stderr, "Error: Stack limit reached, not enough memory to
create an integer\n"); return -1;
127         }
128         if(frameStatusList[frame_counter - 1].frameaddress - (top_StackFrame
- sizeof(int)) > MAX_FRAME_SIZE){
129             fprintf(stderr, "Error: Frame is full, cannot create more data on
it\n"); return -1;
130         }
131     }
132     else if(var_type == 2){
133         if(top_StackFrame - sizeof(double) < 300){
134             fprintf(stderr, "Error: Stack limit reached, not enough memory to
create a double\n"); return -1;
135         }
136         if(frameStatusList[frame_counter - 1].frameaddress - (top_StackFrame
- sizeof(double)) > MAX_FRAME_SIZE){
137             fprintf(stderr, "Error: Frame is full, cannot create more data on
it\n"); return -1;
138         }
139     }
140     else if(var_type == 3){
141         if(top_StackFrame - sizeof(char) < 300){
142             fprintf(stderr, "Error: Stack limit reached, not enough memory to
create a char\n"); return -1;
143         }
144     }
```

```
144     if(frameStatusList[frame_counter - 1].frameaddress - (top_StackFrame
145         - sizeof(char)) > MAX_FRAME_SIZE){
146         fprintf(stderr, "Error: Frame is full, cannot create more data on
147         it\n"); return -1;
148     }
149     } return 0;
150 }
151 void CF(char* functionname, int functionaddress){
152     if(strlen(functionname) > 8){
153         fprintf(stderr, "Error: Function name exceeds 8 characters\n");
154         return;
155     }
156     if(MEMSIZE - (top_StackFrame + 1) + MIN_FRAME_SIZE > MAX_STACK_SIZE){
157         fprintf(stderr, "Error: Stack Overflow, not enough memory available
158         for new function\n"); return;
159     }
160     if(frame_counter == MAX_FRAMES){
161         fprintf(stderr, "Error: Cannot create another frame, maximum number
162         of frames have been reached\n"); return;
163     }
164     for(int i = 0; i < frame_counter; i++){
165         if(strcmp(frameStatusList[i].name, functionname) == 0){
166             fprintf(stderr, "Error: Function already exists with the given name
167             \n"); return;
168         }
169     }
170     frameStatusList[frame_counter].used = true;
171     frameStatusList[frame_counter].number = frame_counter;
172     strncpy(frameStatusList[frame_counter].name, functionname, sizeof(
173         frameStatusList[frame_counter].name));
174     if(frame_counter > 0 && ((frameStatusList[frame_counter - 1].
175         frameaddress - top_StackFrame) < 10)) top_StackFrame = frameStatusList
176         [frame_counter - 1].frameaddress - 10;
177     frameStatusList[frame_counter].functionaddress = functionaddress;
178     frameStatusList[frame_counter].frameaddress = top_StackFrame;
179     memcpy(&memory[394], &frameStatusList, sizeof(frameStatusList));
180     frame_counter++;
181     printf("Created frame %s with address %d\n", functionname,
182         functionaddress); return;
183 }
184 void DF(){
185     if(frame_counter == 0){
186         fprintf(stderr, "Error: Stack is empty, no frames to delete\n");
187         return;
188     }
189     for(int i = top_StackFrame; i < frameStatusList[frame_counter - 1].
190         frameaddress; i++) memory[i] = 0;
191     if(frame_counter == 1) top_StackFrame = 393;
192     else top_StackFrame = frameStatusList[frame_counter - 1].frameaddress;
193     frame_counter--;
194     frameStatusList[frame_counter] = (struct framestatus){0};
```

```
185 memcpy(&memory[394], &frameStatusList, sizeof(frameStatusList));
186 printf("Deleted Frame\n"); return;
187 }
188
189 void CI(char* integername, int integervalue){
190     if(var_errors(1) == -1) return;
191     top_StackFrame -= sizeof(int);
192     memcpy(&memory[top_StackFrame], &integervalue, sizeof(int));
193     printf("Created integer %s with value %d\n", integername, integervalue);
194     return;
195 }
196
197 void CD(char* doublename, double doublevalue){
198     if(var_errors(2) == -1) return;
199     top_StackFrame -= sizeof(double);
200     memcpy(&memory[top_StackFrame], &doublevalue, sizeof(double));
201     printf("Created double %s with value %lf\n", doublename, doublevalue);
202     return;
203 }
204
205 void CC(char* charname, char charvalue){
206     if(var_errors(3) == -1) return;
207     top_StackFrame -= sizeof(char);
208     memcpy(&memory[top_StackFrame], &charvalue, sizeof(char));
209     printf("Created char %s with value %c\n", charname, charvalue); return;
210 }
211
212 void CH(char* buffername, int size){
213     if(size <= 0){
214         fprintf(stderr, "Error: Size of buffer cannot be less than or equal
215         to 0\n"); return;
216     }
217     int total_size = size + BUFFER_METADATA_SIZE;
218
219     if(frame_counter == 0){
220         fprintf(stderr, "Error: Stack is empty, no frames to create pointer
221         to buffer in\n"); return;
222     }
223     if(top_StackFrame - sizeof(int) < 300){
224         fprintf(stderr, "Error: Stack limit reached, not enough memory to
225         create a pointer to buffer\n"); return;
226     }
227     if(frameStatusList[frame_counter - 1].frameaddress - (top_StackFrame -
228     sizeof(int)) > 80){
229         fprintf(stderr, "Error: Frame is full, cannot create more data on it\
230         n"); return;
231     }
232
233     struct freelist* curr = fl_head;
234     while(curr != NULL && curr->size < total_size) curr = curr->next;
235     if(curr == NULL){
236         fprintf(stderr, "Error: The heap is full, cannot create more data on
237         it\n"); return;
238     }
```

```
230 }
231 int magic = rand(), heapStart = curr->start;
232 memcpy(&memory[heapStart], &size, sizeof(int));
233 memcpy(&memory[heapStart + sizeof(int)], &magic, sizeof(int));
234 for(int i = heapStart + BUFFER_METADATA_SIZE; i < heapStart +
    total_size; i++) memory[i] = 'a' + (rand() % 26);
235
236 curr->start += total_size;
237 curr->size -= total_size;
238
239 struct allocated* newAlloc = (struct allocated*) malloc(sizeof(struct
    allocated));
240 strcpy(newAlloc->name, buffername);
241 newAlloc->startaddress = heapStart;
242 newAlloc->next = a_head->next;
243 a_head->next = newAlloc;
244
245 top_StackFrame -= sizeof(int);
246 int bufferAddress = heapStart;
247 memcpy(&memory[top_StackFrame], &bufferAddress, sizeof(int));
248
249 printf("Created buffer %s with size %d\n", buffername, size); return;
250 }
251
252 void DH(char* buffername){
253     if(a_head == NULL){
254         fprintf(stderr, "Error: The pointer is NULL or already de-allocated\n
    "); return;
255     }
256     struct allocated* prev = NULL, *curr = a_head;
257     while(curr != NULL && strcmp(curr->name, buffername) != 0){
258         prev = curr; curr = curr->next;
259     }
260     if(curr == NULL){
261         fprintf(stderr, "Error: The pointer is NULL or already de-allocated\n
    "); return;
262     }
263     int bufferSize = *(int*)(memory + curr->startaddress);
264     memset(memory + curr->startaddress, 0, bufferSize +
    BUFFER_METADATA_SIZE);
265
266     if(prev != NULL) prev->next = curr->next;
267     else a_head = curr->next;
268
269     struct freelist* newFree = (struct freelist*) malloc(sizeof(struct
    freelist));
270     newFree->start = curr->startaddress;
271     newFree->size = bufferSize + BUFFER_METADATA_SIZE;
272     newFree->next = fl_head;
273     fl_head = newFree;
274
275     free(curr);
276 }
```



```
277 printf("De-allocated buffer %s\n", buffername); return;
278 }
279
280 void printFreeList(){
281     struct freelist* curr = fl_head;
282     printf("Free List: \n");
283     while(curr != NULL){
284         printf("Start: %d, Size: %d\n", curr->start, curr->size);
285         curr = curr->next;
286     }
287     printf("\n");
288 }
289
290 void printAllocatedList(){
291     struct allocated* curr = a_head->next;
292     printf("Allocated List: \n");
293     while(curr != NULL){
294         printf("Name: %s, Start Address: %d\n", curr->name, curr->
startaddress);
295         curr = curr->next;
296     }
297     printf("\n");
298 }
299
300 void SM(){
301     for(int i = MEMSIZE; i >= 0; i--){
302         printf("%d: %X \n", i, memory[i]);
303     }
304     printFreeList(); printAllocatedList();
305
306     printf("
#-----#\n");
307 }
```

Listing 12: Stack and Heap Memory Management Program