Habib University
Operating Systems - CS232
Assignment 02 - Report

Instructor: Muhammad Mobeen Movania
Dua Batool - db07098

Contents

1 General Structure

1.1 Memory

The total memory is 500 bytes, and it is a character array. Memory locations 395 to 500 are reserved for stack frames metadata. The stack is initially 200 bytes, starting from memory location 300 and going up to 500. It can grow until memory location 255.
The heap is initially 100 bytes, starting from memory location 0 and going up to 100. It can grow until memory location 255.

1.2 Structures

1.2.1 Freelist

The freelist has a linked list data structure. It starts initially from memory location 0 and it has size 100 since initially there is no data in heap. Its start address and size are updated as data is inserted into the heap. If there are multiple buffers on heap and a buffer is deleted from the middle, a new node of the freelist is created.

1.2.2 Allocated List

The allocated has a linked list data structure. It updates as data is added to the heap. It stores the name of the buffer, the address in the stack where the stack pointer is stored, and the next allocated buffer. The stack pointer is used to keep track of the memory location of the heap where the buffer is stored.

1.3 Global Variables

There is only one global variable, `top_of_stack`, which is used to keep track of the top of the stack. This variable is useful in knowing if the `currentstacksize` is supposed to be increased or not, as well as helpful in keeping track of the size of the current frame so that the total bytes of the current frame does not exceed 80 bytes.

2 Functions

2.1 Helper Functions

- **SetNull**: It is used to initialize the values of all stack frames before they are used in the functions. It also sets the value of each byte of memory to `\0`.
- **GetFrame**: It is used to get the index when a new frame is being created. It returns -1 when all 5 frames are in use.
- **CurrentFrame**: It is used to get the index of the current frame that is in use. It returns -1 when there is no frame on stack.

2.2 CreateFrame

void CreateFrame(all_frames * fs, int * currentstacksize, char name[], int address)
This function gets the index to create a new frame. After the error handling, it updates the `framestatus` accordingly and allocates 10 bytes on stack for this frame. It uses `all_frames` to

access the `framestats` of the index returned by the helper function.

Error Handling:

- **Name Check**: Gives an error if a function with the same name as given in the `CreateFrame` argument already exists.
- **Frames Check**: Gives an error if there are already 5 frames on the stack, since there is no space to create another frame.
- **Stack Check**: Gives in error if stack is utilized till memory location 255, and creating a frame will require the stack to go beyond this location.

2.3 DeleteFrame

void DeleteFrame(all_frames * fs, char memory[], char occupied[], allocated * allocated_region)
This function gets the index of the frame to be deleted. It then sets the memory occupied by this frame to `\0`. It updates the `top_of_stack` to the memory location of the `frameaddress` of this function, as well as initializes the `framestatus` of this frame so that this index could be used when another frame is being created.

Error Handling:

- **Stack Empty**: Gives an error if user is attempting to delete a frame when no frame exists.

2.4 CreateIntVar, CreateDoubleVar, CreateCharVar

These three functions majorly have the same functionality except for one part that the data type which is inserted in the memory is different.

- **void CreateIntVar(all_frames * fs, int * currentstacksize, char memory[], char occupied[], char name[], int value)**
- **void CreateDoubleVar(all_frames * fs, int * currentstacksize, char memory[], char occupied[], char name[], double value)**
- **void CreateCharVar(all_frames * fs, int * currentstacksize, char memory[], char occupied[], char name[], char value)**

These functions get the index of the current frame and add data into the memory location of the respective frame. For integers, 4 bytes are allocated, for doubles, 8 bytes are allocated, and for characters, 1 byte is allocated. These bytes in the memory are casted into the data types of the data being stored. Frame size is increased if there is not enough space on the frame and `currentstacksize` is increased if more stack space is needed.

Error Handling:

- **No frame**: Gives an error if user attempts to insert data without any frame existing on stack.
- **Frame full**: Gives an error if user attempts to insert data when the frame is full.
- **Stack Overflow**: Gives an error if the maximum memory allocated for stack is not enough to insert data.

2.5 CreateBuffer

void CreateBuffer(all_frames * fs, char memory[], char occupied[], int * heapsize, allocated * new_node, allocated * allocated_region, freelist * free_region, char buffername[], int buffersize)
This function allocates 4 bytes in the memory space allocated for the current stack frame to store the address of the heap memory location. It creates a buffer of size `buffersize` as given by the user and 8 bytes in addition for that buffer (for magic number). Then it fills the heap memory allocated for the buffer (excluding the 8 bytes) with random characters.

`allocated_region` is the allocated_list. A new node is added to this list when a buffer is created. `free_region` is the freelist. It is updated according to the size of the buffer.

Error Handling:

- **No Frame**: Gives an error if there is no frame on stack. Need a frame to create a stack pointer which will be pointing towards the memory location in heap.
- **Frame Full**: Gives an error if there is no space on the current frame to allocate bytes for the stack pointer.

2.6 RemoveBuffer

void RemoveBuffer(all_frames * fs, char memory[], char occupied[], int * currentheapsize, freelist * free_node, allocated * allocated_region, freelist * free_region, char name[])
This function updates the allocated list by removing the node of the buffer to be deleted. It also updates the free list, a new node is added in the free list if a buffer being removed is in the middle of other buffers.

It does not deallocate the 4 bytes in the stack memory that are pointing towards the heap memory, which results in that stack pointer becoming a dangling pointer.

Error Handling:

- **Buffer not found**: Gives an error if the user attempts to delete a buffer which does not exist.

2.7 ShowMemory

void ShowMemory(all_frames * fs, char memory[], char occupied[], int currentstacksize, int currentheapsize)
Shows the following things in memory:

- The memory location on which the `framestatus` of the 5 frames are reserved.
- The start and end of memory locations of stack (excluding the locations of `framestatus`. It then shows the values stored in the memory, including the location on the memory where they are store, and their datatype.
- The start and end of memory locations of heap. It prints the contents of the buffer(s) if any buffer is allocated on heap.

2.8 Main

int main (int argc, char * argv[]):
The main function defines the arrays (`memory, occupied`), the structures (`all_frames,freelist,allocated`). It calls the helper function `SetNull` to initialize their values then creates a shell like environment. It keeps taking input from the user until the user enters `exit` and calls the respective functions according the the commands user entered.

3 Input

There is a shell like environment where user can enter commands and the respective functions will be called. The user has to enter `exit` to end the program.

Error Handling:

- **Number of Arguments**: Gives an error if the user does not provide with the correct number of arguments for the command entered.
- **Incorrect Command**: Gives an error if an incorrect command has been entered.

4 Printing

I have allocated an array `occupied` of 500 bytes, just so we can clearly visualize what type of data is stored where in the memory. The code will work completely fine even if this array is removed, it is just there for a good snapshot of the memory.