Report

The project involves creating a wrapper for mmap. However, in this context, the requests to mmap are limited to multiples of the page size. The primary goal here is to establish a free-list component within the process that manages both empty memory (holes) and occupied memory. The virtual address provided by mmap is considered a physical address and needs to be converted to a virtual address for the particular process, starting from 0.

A standard C project folder structure was followed placing appropriate files in respective folders. mems.h and dll.h are located in the include folder.

“dll.h”

This header file, located in the include folder, contains the node structure for the freelist and the returnNewAddr() function to facilitate functioning of the FreeList for the MeMS.

struct node

struct node consists of the following attributes
void* pAllocAddress; - contains the address of the memory allocated upon user request.
_Bool isSub; - to mark whether the node is part of a sublist.
struct node* pSubChain; - contains the address to the first node of the sublist.
long size; - size of the memory allocated upon invoking mems_alloc().
struct node* pPrev;
struct node* pNext;
_Bool isHole;
_Bool deleted; - to mark the node is deleted, crucial for reinserting a new node in the same slot (explained in detail below).

returnNewAddr()

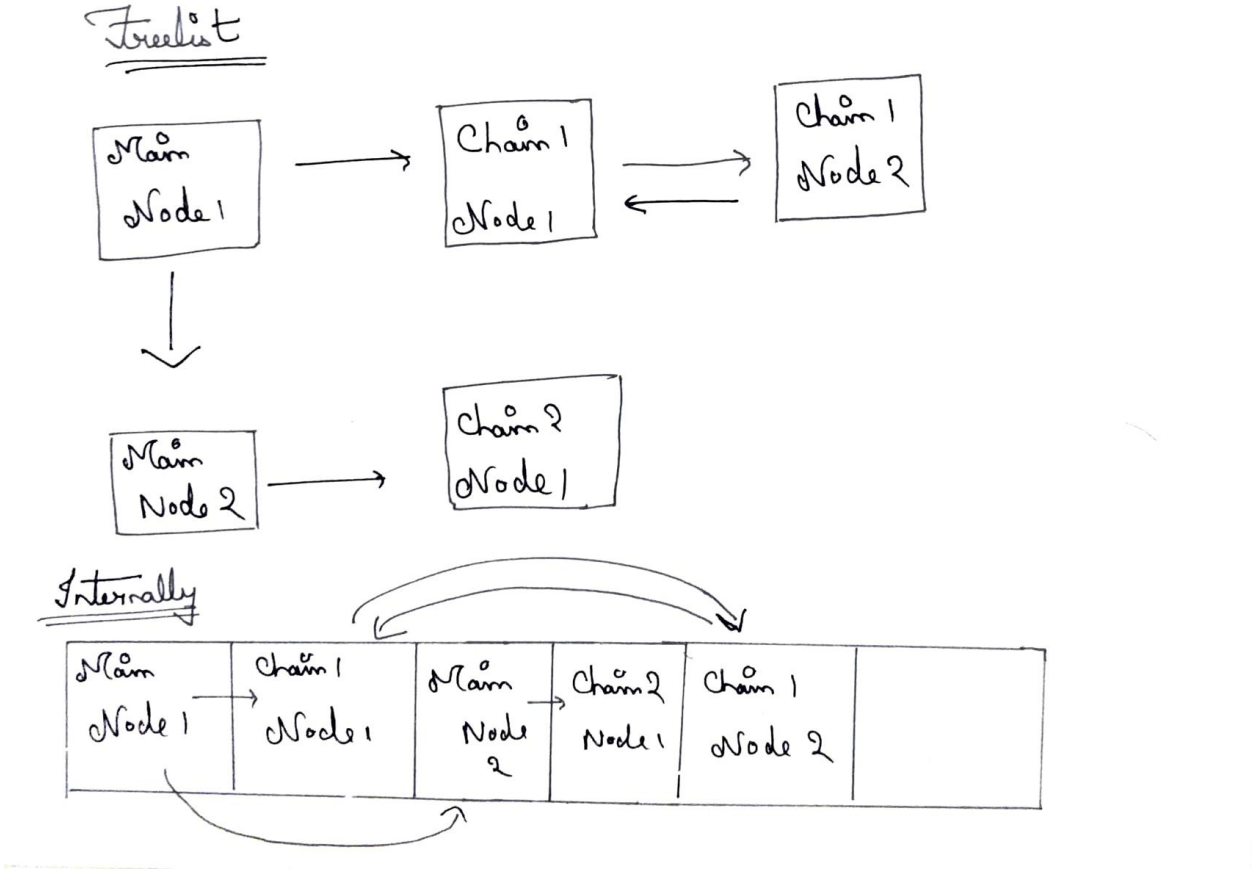
To achieve a fully functioning free list, we must first decide on the data structure. For the sake of convenience, we are using a doubly linked list for both the main list and each respective sublist. Since there is not much difference between the nodes in these two lists, the free list can be constructed using the same node structure.

However, the challenge lies in how to create a free list without using malloc. Typically, malloc is commonly used when implementing data structures in C. In this case, we fulfill this requirement using the mmap system call. Given that we can only invoke mmap to allocate memory in multiples of PAGE_SIZE, we initially allocate one page for the free list and cast the return address to (struct node*), where struct node represents the node structure created for each individual node in the free list. By typecasting the allocated memory return address and ensuring that the size of a struct node is less than PAGE_SIZE, we obtain an array of struct nodes, containing PAGE_SIZE / sizeof(struct node) elements.

The approach here involves creating a returnNewAddr() function that, when invoked, finds an empty slot in the array for inserting a new node. If the array is full, the function increases the array size by adding another page to the original size, copies all old elements into the new one, and returns the first index after it. The idea is that regardless of how the list nodes might be linked to one another, internally each individual node will be stored in the array at any index location.

This is how we were able to create a system for a properly functioning free list. Every time we want to insert a node into the list, we invoke returnNewAdd(), which searches for an empty slot in the array, returns the address, and then constructs the node attributes and links them to other nodes accordingly. We only store the address for the main head node.

Additionally, it's important to note that the first node in this free list is treated as a dummy node for coding logic convenience.



mems_alloc()

Now that we have a fully functioning free list system, we can develop logic for memory allocation using mems_alloc(). The logic is straightforward. When the function is invoked, requesting a specific memory size, we traverse the sub-list nodes of each main node. If we find a hole with a size less than the requested size, we split the node into two nodes: one of type process with the requested size and another of type hole containing the remainder size.

In case there are no nodes of type hole with size less than or equal to the requested size, we create a new main node. This new main node would have a size equal to ceil(size of memory requested/PAGE_SIZE) * PAGE_SIZE. The sublist would contain two nodes: one of type process with the requested size and another of type hole containing the remainder size, if any.

We have assumed that virtual addresses returned by MeMS for the process's Assumed Virtual Address Space start from 0. The address is calculated by

summing up the sizes of all nodes up to the very node whose pointer to the memory is being returned and then casting it into (void*).

mems_get()

For mems_get(), we reverse the process used in mems_alloc(). Instead of summing up the sizes of nodes, we take the virtual address and iterate through all main nodes. At each iteration, we subtract the size from the virtual address. If, at a particular main node, the virtual address is less than the node's size, we enter the sublist; otherwise, we move to the next node, subtracting the size from the virtual address. Within each sublist, only when we encounter a node with a size greater than the virtual address, we return the node's allocated address plus the virtual address.

However, there can be multiple error scenarios to handle. If the virtual address corresponds to a node of type hole, we return an error, writing the appropriate message into perror() and returning (void*) -1. If the virtual address provided is negative, we return the same error. If the address exceeds the size of the sum of all nodes, it's considered an invalid memory address, and we return an error accordingly.

mems_print_stats()

The mems_print_stats() function involves iterating through each main node and each subnode and printing relevant details.

mems_free()

For mems_free(), we use the same algorithm for searching nodes as in mems_get and mark the particular node as isHole = true. Then, we check through the list for consecutive nodes of type hole and merge them. The size of the second node is added to the first one, and the second node is marked as deleted in the free list, making that slot in the array available for future insertion of a new node.