(MeMS) Memory Management System

Before starting the assignment clone the skeleton code for the assignment from the below github repo:

https://github.com/Rahul-Agrawal-09/MeMS-Skeleton-code.git

Problem Statement:

Implement a custom memory management system (MeMS) using the C programming language. MeMS should utilize the system calls **mmap** and **munmap** for memory allocation and deallocation, respectively. The system must satisfy the following constraints and requirements outlined below:

Constraints and Requirements:

MeMS can solely use the system calls **mmap** and **munmap** for memory management. The use of any other memory management library functions such as **malloc**, **calloc**, **free**, and **realloc** are **STRICTLY PROHIBITED**.

MeMS should request memory from the OS using **mmap** in multiples of the system's **PAGE_SIZE**, which can be determined using the command **getconf PAGE_SIZE**. For most Linux distributions, the **PAGE_SIZE** is 4096 bytes (4 KB); however, it might differ for other systems.

MeMS should deallocate memory only through **munmap** and deallocation should only occur in multiples of **PAGE_SIZE**.

As the value of **PAGE_SIZE** can differ from system to system hence use the macro "PAGE_SIZE" provided in the template wherever you need the value of PAGE_SIZE in your code so that this size can be modified if required for evaluation purposes.

The user program must use the functions provided by MeMS for memory allocation and deallocation. It is not allowed to use any other memory management library functions, including **malloc**, **calloc**, **free**, **realloc**, **mmap**, and **munmap**.

Although MeMS requests memory from the OS in multiples of PAGE_SIZE, it only allocates that much amount of memory to the user program as requested by the user program. MeMS maintains a free list data structure to keep track of the heap memory which MeMS has requested from the OS. This free list keeps track of two items:

memory allocated to each user program. We will call this memory as PROCESS in the free list (details below).

Memory which has not been allocated to any user program. We will call this memory as a HOLE in the free list (details below).

Free List Structure:

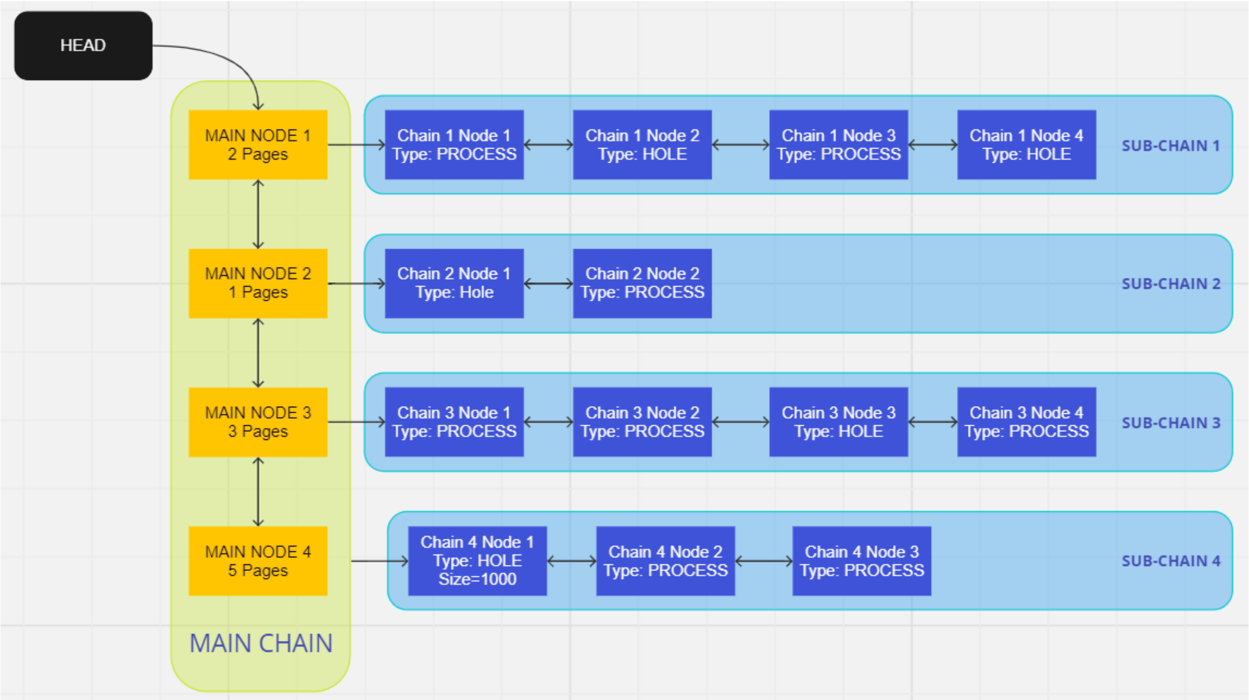
Free List is represented as a doubly linked list. Let's call this doubly linked list as the **main chain** of the free list. The main features of the main chain are:

Whenever MeMS requests memory from the OS (using mmap), it adds a new node to the main chain.

Each node of the main chain points to another doubly linked list which we call as **sub-chain**. This sub-chain can contain multiple nodes. Each node corresponds to a segment of memory within the range of the memory defined by its main chain node. Some of these nodes (segments) in the sub-chain are mapped to the user program. We call such nodes (segments) as **PROCESS** nodes. Rest of the nodes in the sub-chain are not mapped to the user program and are called as **HOLES** or **HOLE** nodes.

Whenever the user program requests for memory from MeMS, MeMS first tries to find a sufficiently large segment in any sub-chain of any node in the main chain. If a sufficiently large segment is found, MeMS uses it to allocate memory to the user program and updates the segment's type from HOLE to PROCESS. Else, MeMS requests the OS to allocate more memory on the heap (using mmap) and add a new node corresponding to it in the main chain.

The structure of free list looks like below:



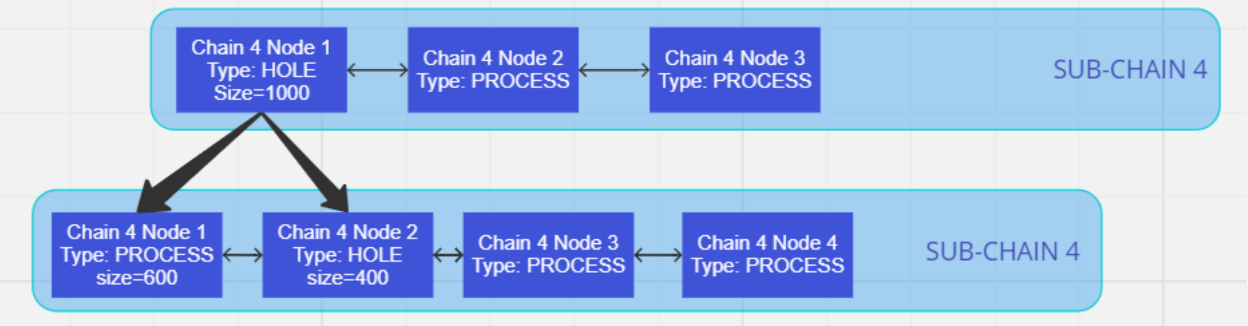
The main features of the chain (sub-chain) are:

Each chain is broken into segments.

Each segment represents a part of memory and represents whether that segment is of type PROCESS i.e. is mapped to the user process or is of type HOLE i.e. not allocated/mapped to the user program.

The segments of type HOLE can be reallocated to any new requests by the user process. In this scenario, if some space remains after allocation then the remaining part becomes a new segment of type HOLE in that sub-chain.

Graphaphically it looks something like below:



In the above picture, the Node1 of sub-chain-4 is reused by the user process but only 600 bytes out of 1000 bytes are used. Hence a HOLE of 400 bytes is created and the node of 600 bytes is marked as PROCESS and the MeMS virtual address corresponding to 600 bytes node is returned to the user process for further use.

NOTE: You must handle the corner cases and make sure that your system should avoids memory fragmentation within the free list.

MeMS Virtual Address and MeMS Physical Address:

Let us call the address (memory location) returned by mmap as the MeMS physical address. In reality, the address returned by mmap is actually a virtual address in the virtual address space of the process in which MeMS is running. But for the sake of this assignment, since we are simulating memory management by the OS, we will call the virtual address returned by mmap as MeMS physical address.

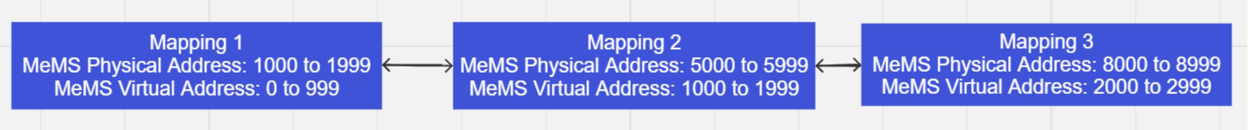
Just like a call to **mmap** returns a virtual address in the virtual address space of the calling process, a call to **mems_malloc** will return a MeMS virtual address in the MeMS virtual address space of the calling process. For the sake of this assignment, MeMS manages heap memory for only one process at a time.

Just like OS maintains a mapping from virtual address space to physical address space, MeMS maintains a mapping from MeMS virtual address space to MeMS physical address space. So, for every MeMS physical address (which is provided by mmap), we need to assign a MeMS virtual address. As you may understand, this MeMS virtual address has no meaning outside the MeMS system.

Any time the user process wants to write/store anything to the heap, it has to make use of the MeMS virtual address. But we cannot directly write using MeMS virtual address as the OS does not have any understanding of MeMS virtual address space. Therefore, we first need to get the MeMS physical address for that MeMS virtual address. Then, the user process needs to use this MeMS physical address to write on the heap.

For example in the below figure

MeMS virtual address 0 corresponds to MeMS physical address 1000
MeMS virtual address 500 corresponds to MeMS physical address 1500
MeMS virtual address 1024 corresponds to MeMS physical address 5024


Mapping 1
MeMS Physical Address: 1000 to 1999
MeMS Virtual Address: 0 to 999
Mapping 2
MeMS Physical Address: 5000 to 5999
MeMS Virtual Address: 1000 to 1999
Mapping 3
MeMS Physical Address: 8000 to 8999
MeMS Virtual Address: 2000 to 2999

We can get the MeMS physical address (i.e. the actual address returned by mmap) corresponding to a MeMS virtual address by using the function mems_get function (see below for more details).

Function Implementations:

Implement the following functions within MeMS:

void mems_init(): Initializes all the required parameters for the MeMS system. The main parameters to be initialized are

the head of the free list i.e. the pointer that points to the head of the free list

the starting MeMS virtual address from which the heap in our MeMS virtual address space will start.

any other global variable that you want for the MeMS implementation can be initialized here.

Input Parameter: Nothing

Returns: Nothing

void mems_finish(): This function will be called at the end of the MeMS system and its main job is to unmap the allocated memory using the munmap system call.

Input Parameter: Nothing

Returns: Nothing

void* mems_malloc(size_t size): Allocates memory of the specified **size** by reusing a segment from the free list if a sufficiently large segment is available. Else, uses the mmap system call to allocate more memory on the heap and updates the free list accordingly.

Parameter: The size of the memory the user program wants

Returns: MeMS Virtual address (that is created by MeMS)

void mems_free(void* ptr): Frees the memory pointed by **ptr** by marking the corresponding sub-chain node in the free list as HOLE. Once a sub-chain node is marked as HOLE, it becomes available for future allocations.

Parameter: MeMS Virtual address (that is created by MeMS)

Returns: nothing

void mems_print_stats(): Prints the total number of mapped pages (using mmap) and the unused memory in bytes (the total size of holes in the free list). It also prints details about each node in the main chain and each segment (PROCESS or HOLE) in the sub-chain.

Parameter: Nothing

Returns: Nothing but should print the necessary information on STDOUT

void *mems_get(void*v_ptr): Returns the MeMS physical address mapped to **ptr (** ptr is MeMS virtual address).

Parameter: MeMS Virtual address (that is created by MeMS)

Returns: MeMS physical address mapped to the passed **ptr** (MeMS virtual address).

Submission Guidelines:

Please download the skeleton template from the below github repository:
https://github.com/Rahul-Agrawal-09/MeMS-Skeleton-code

Students are required to submit the C code containing the implementation of all the functions there in the skeleton template provided, along with a detailed explanation of their approach. Additionally, students should provide a demonstration of their implementation with test cases that validate the functionality of MeMS under different scenarios.

(Note: Do not change any function's name or signature provided in the skeleton template. You are free to include more files if you want.)

Some more guidelines [Added on October 28th, 2023]

You will be graded upon:

- Documentation
- Viva/Demo
- Error handling
- Covering all the edge cases (like joining 2 adjacent hole nodes to make a new hole node)
- Correct mems_virtual_address corresponding to mems_physical_address.
- We will run various test cases (variants of the example file which we have shared) and each of them will have individual weightage.
- Successful compilation and correct output format.
- Use the data structures which have been described in the documentation.
- Any other implementation specific detail necessary for the assignment.

Note:

1. Use the PAGE_SIZE macro in the code so that we can test your code with different page size values.
2. Use Github for version control as TA's might check commit log

Suggestion:

1. Maintain good program structure and proper naming convention so that evaluation can be done without any hassle