Report for Assignment #2

# Jose Toledo and Jasmine Batten

## Functions and their Usage

* Struct node holds an void pointer to an address, its size (in memory), and a pointer to another node called next. It allows us to implement a linked list.
* Struct list keeps track of the linked list for the user’s memory and holds a node pointer called head and an unsigned long variable for the amount of remainingMemory that can be allocated.
* Mem\_init takes a size and a policy as input and sets a global variable called initFlag to 1 and also sets a global variable called PolicySet to the policy the user entered. Then it finds the page size by checking to make sure that the initial size is between 0 and the getpagesize(). Otherwise, we round the page size up so we have an integer number of page sizes. Finally, we create a file descriptor called /dev/zero that we will use to “allocate” a space in memory for us to use using mmap(). /dev/zero essentially will fill that space in memory with zeroes so it is clean and able to be filled with our own data. We also set a limit for us to use later, meaning how much space in memory we have to use, by finding the original address and adding the number of pages we have allocated. This function will return -1 if the mapping fails or the memory was already initialized, or will return 0 if it returns successfully.
* \*Mem\_alloc takes a size as an input and first checks to make sure that memory was initialized and will return NULL if it wasn’t. Then we utilize a linked list in order to maintain a list of nodes which contain pointers to their addresses, their sizes, and a pointer to the node in front of them. The list of nodes includes a node pointer and the remainingMemory left in the block. Then we create new nodes called curr and next, and then we assign curr to be the head of the memory list and next to be curr’s next pointer. Please note that this space in memory is used for the linked list only and not for the user’s data. This is so we can save the space the linked list takes up in memory instead of using malloc().Then it will check all of the policies individually depending on what the user set the memory to be, including first fit, best fit, and worst fit.
  + FIRST-FIT POLICY: We next check to see if the policy set is the first fit policy, meaning that the space will be allocated the first time it finds a space large enough to hold the memory node. If it is, we map a new space in memory (and fill it with zeroes using /dev/zero) for the user’s new data node (called temp) and make it specifically that size. *Note that the point of mapping a new space in memory for the temporary node allows us to hide the linked list data structure from the user while maintaining the linked list’s overhead in memory and also allows us to link the user’s data nodes with the linked list nodes.*
    - We check if it will fit at the head of the list, between nodes, and at the end of the list and place the node where it fits first. We also update any linked list information and hide this information from the user by simply returning the address of the node they placed.
    - If it reaches the end without placing the node, it uses munmap() to deallocate the space we allocated at the beginning and returns NULL to indicate there was no space found.
  + BEST-FIT POLICY: We create any linked list nodes we might need and a new new unsigned long for the smallestSpace we have found throughout the memory. Again we map space to the new node being added.
    - First we go through the entire memory block and check to see which part of memory contains the smallestSpace.
    - If the best fit is either in the beginning, in between nodes, or at the end, we place the node where it fits best and we update any linked list information and hide this information from the user by simply returning the address of the node they placed.
    - If none of the options were used, then there was no space found and the temp node is deallocated using munmap() and the function returns NULL.
  + WORST-FIT POLICY: We create any linked list nodes we might need and a new unsigned long for the largestSpace we have found throughout the memory. Again we map space to the new node being added using mmap().
    - First we go through the entire memory block and check to see which part of memory contains the largestSpace.
    - If the worst is either in the beginning, in between nodes, or at the end, we place the node where it fits the worst and we update any linked list information. We hide this information from the user by simply returning the address of the node they placed.
    - If none of the options were used, then there was no space found and the temp node is deallocated using munmap() and the function returns NULL.
* Mem\_Free takes a pointer and goes through the linked list structure and checks each node to see if an object exists there. If it does exist and it’s the head of the linked list, it will update the linked list by changing the pointers of the head of the linked list to the next node in the list and then using munmap() to deallocate the node and will return 0. If it’s the only node in the linked list, it will deallocate the entire block of memory and return 0. If the pointer is within the nodes that are allocated (it’s not the head), it will update the linked list to point the previous node’s next pointer to the currently removed node’s next node and will deallocate that node using munmap() and will return 0. If it doesn’t satisfy either of these conditions, it returns -1.
* Mem\_IsValid goes through the linked list and compares the start and end addresses of the current node and checks to see if the pointer given falls within that range. If it does then it returns 1, if it doesn’t it returns 0.
* Mem\_GetSize goes through the linked list and compares the start and end addresses of the current node and checks to see if the pointer given falls within that range. If it does, it gets the size of the current node and returns it. If the node isn’t found, it returns -1.
* Mem\_GetFragmentation goes through the linked list and finds the largest block of free space and all the free space within the block. Then it returns the fragmentation factor, which is the largest free chunk divided by the total remaining memory. It returns 1 if there is no free space in the block.

## Work done by team members

* We discussed the initial data structures we would use for the project and how to store the data structure in the memory while maintaining the data structure for the user, and any variables we would need. After we had a basic understanding, we both worked separately on the assignment and merged our code at the end. We discussed any bugs or issues we may have run into and why they occurred. We worked on the assignment separately at first so that we could learn how to work on the assignment independently and also come together to fix any errors and discuss it to have a better understanding of choices we made.