INDIAN INSTITUTE OF TECHNOLOGY
KHARAGPUR

OS Lab Assignment 5

Memory Management Report

Group 6

Gaurav Madkaikar (19CS30018)

Girish Kumar (19CS30019)

a. What is the structure of your internal page table? Why?

- The internal page table is maintained as a **linked list** with 2 reference pointers ***pageTable** (head) and ***currPageTable** (tail). The fields corresponding to each node of this list are shown below:

```
// Linked List for pageTable
typedef struct PTnode
{
   int localAddress;          // Local address (counter value * 4)
   char *name;                // Name of the variable
   char *type;                // Data type of the variable
   unsigned int size;         // Allocated size
   int functionScope;         // Value of function counter
   void *phyAddress;          // Address in physical memory
   bool listType;             // Check if the entry is a list or not
   bool valid;                // Check if it is used/unused
   struct PTnode *next;       // Next pointer
} listNode;

listNode *pageTable, *currPageTable;
```

This internal page table will help us in mapping local variables to their actual physical address spaces. We have chosen to implement this symbol table as a linked list due to its efficiency and ease in implementation. A possible implementation would have been the use of hash tables by having variable names as keys and the corresponding physical addresses as attributes, however, due to possible deletions of entries and existence of memory holes, we preferred the linked list implementation.

We have also maintained an additional stack **pageTableStack** which stores the addresses freed up due to deletion of **pageTable** entries. By using a greedy strategy, we allocate these addresses (if available) to future variables.

<u>Worst Case Time Complexity Analysis</u>
Insertion - O(1)
Deletion - O(N)

b. What are additional data structures/functions used in your library? Describe all with justifications.

- Additional data structures that we have used include:
  1) Two stacks: **globalStack** and **pageTableStack**
     globalStack: Stores all the local variables in the current function scope which can later be freed.
     pageTableStack: Stores local logical memory addresses along with some additional information which helps in managing memory holes within the pageTable.
  2) A static list of maximum size 100 to store the free memory addresses (along with some printing information) in the physical memory: **freeMemList**
  3) Two global boolean variables: **breakSignal** and **flushGarbage** indicating when **gc_run** can free elements from the pageTable as well as the physical memory.

d. What is your logic for running compact in Garbage collection, why?

- We use a greedy strategy to allocate **pageTable** entries as well as elements in the physical address using two structures: **freeMemList** and **pageTableStack**. These structures are filled by the **gc_run** function when any variable is freed from the memory. While declaring a new variable through the use of **createVar(),** we check if the newly allocated variable can be accommodated within the free space left out in these structures and always allocate it if possible. Since we are directly allocating free memory to a new variable, we are able to save a lot of overhead involved in pushing all variables down in the physical memory upwards. The code can be checked for better reference.

e. Did you use locks in your library? Why or why not?

- We have used locks only at one part of the code, i.e. when the **gc_run** function is activated by the **flushGarbage** variable to free unused memory. This **flushGarbage** variable is activated at the end of the **freeElem()** function. The steps involved in freeing memory within the **gc_run** function are as follows:
    1) Put an entry into the **freeElemList**
    2) Remove the element from the global stack
    3) Remove the element from the pageTable
    4) Initialize the physical address to **NULL**

To prevent concurrent access to these structures during removal of elements which may result in a segmentation fault, we have used only one lock within this runner method.