Operating Systems Laboratory (CS39002)
Spring 2022

Assignment - 05

Hands-on experience for creating better memory management systems

1) ***What is the structure of your internal page table? Why?*** **:**

- ➢ Our page table is a contiguous block of memory. The i^{th} page in the page table maps to a variable and its local address.
- ➢ The page table is the data structure used to store the mapping between the virtual addresses and the physical addresses.
- ➢ The page table is maintained as an array of page table entries allocated dynamically. Each entry consists of a name (character array) and an address (int, physical address of the variable).
- ➢ We can then find the physical address by multiplying the local address by 4 (as memory can be accessed in chunks of 4 only).
- ➢ We use this structure to implement the page table so that we can easily extract the physical address from the counter.

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

The data structures used are as follows :

- ➢ symbolTableEntry - It represents an element of our symbol table. It contains all details regarding a variable. This structure is used as an element in the ste array (symbol table).

The structure contains following items:

- *type* - An integer to store type of the variable.
  - 1 - Integer
  - 2 - Character
  - 3 - Medium Integer
  - 4 - Boolean
- *size* - An integer to store the size of the element.
  - 1 for a variable
  - Number of elements in the array for arrays.
- *localCounter* - An integer to store the value of the totalCounter at the time of creation of variable, with the help of this we can find the logical address associated with the variable.
- *mark* - An integer to indicate marking (0 - not marked, 1 - marked) for sweeping phase.
- *name* - Character array to store the name of the variable. Used to find whether an element is in the page table or not.

➢ pageTableEntry - It represents an element of our page table. The purpose of this element is to map the logical address to the local address of the segment which stores the value of that variable. We can then find the physical address by multiplying the local address by 4 (as memory can be accessed in chunks of 4 only). This structure is used as an element in the pte array. The structure contains following items:

- *localAddress* - An integer to store the value of the local address of a variable or array element, with the help of this we can find the physical address associated with it and can access its value in the memory.

- *name* - Character array to store the name of the variable.

➢ *stack* - This structure (implemented as a stack) is used to keep track of the variable references in our user space code. When a function returns we will pop the variables used from the stack by calling a function in our library. The structure contains following items:

- *count* - An integer to store the number of entries in the stack at any moment (variable).
- *name* - A 2D character array to store the names of variables in use in the main program, if any variable scope ends (or we explicitly clears it) we need to pop that element from the stack to identify its end.

➢ *freeSegment* - This structure is used to keep track of the frames or segments (in chunks of 4 bytes) in the memory which are used to store the variables. This structure is used as an element in the freeSegments array. The structure contains following items:

- *count* - An integer to store the number of frames or segments (free 4 byte chunks) free at any given moment.
- *array* - An integer array of static size where i^{th} element denotes the local address of i^{th} free frame or segment of 4 byte.

Our library file contains the following functions:

➢ init_stack - The function takes the size of the stack as input. The function does the following job:

- Allocates memory for the stack
- It initialises the count value of stack to 0.

- push - The function takes a variable name to push into the stack as parameter (char*). The function does the following job:
    - Inserts the variable in the stack (its name).
    - Increments the count value for stack.
- top - The function takes a char pointer as a parameter and copies the name of the top element to the parameter received.
- pop - This function returns the top element of the stack, removes it from the stack and decreases the count of the number of elements in the stack.
- *search_symbolTableEntry* - This function will search the symbol table for the variable (or array) using name as parameter and if found will return its index in the table. If not found it will return -1 denoting that no such variable exists.
- *insert_symbolTableEntry* - This function is used to insert a variable or array into the symbol table. It will take all information regarding the variable as input parameters and then will find the first empty slot in the table. It then inserts the variable to that slot.
- *sweepPhase* - The function takes the index to the variable or array in the symbol table as input. The function does the following job:
    - Resets (removes) the variable from symbolTableEntry because the variable is to be deleted, unmarks the variable because now it is sweeped from our memory.
    - It then checks if the variable name is of array or not by checking size of the variable.
        - If name is not of array then it frees the frame or segment allocated for this variable and also empties the page table entry for this variable.

        - If the name is an array then we first find the size of that array and then free all the segments and empty all the page table entries given to that array.

➢ *compactPhase* - The function does the following job:
    - It will Iterate over the page table and try to find an empty hole. If a hole is found it then marks the found position.
    - For any page after the found position tries to move it as up in the page table as possible in order to fill all the holes.
    - If the variable that we encounter is an array then we will move all the elements of that variable up by the number of holes that we have in the page table from the start.
    - At the end we decrease the localcounter by 4*number of holes encountered during compaction, this way all the holes are aggregated at the end and then discarded.

➢ *gc_run* - This function iterates through every entry in the symbol table, for every variable that is marked for sweeping during freeElem, the function calls sweepPhase with that symbol table index. It then calls compactPhase for compaction because now every variable that is of no use is sweeped.

➢ *gc_initialize* - This is the function of the garbage collector thread.It runs in an infinite loop, in each iteration it sleeps for 1 milliseconds and then calls gc_run for garbage collection. It will come out of the infinite loop once our program ends.

➢ *getVar* - This function takes the name of the variable for which the user wants the value as parameter.

- It will then search the symbol table for the variable. If not found it will print a message that no such variable exists.
- It then checks if the variable is an array, if it is it returns -1 with a message.
- It then extracts the logical address from the symbol table of the variable, using that we will find the local address from the page table of the memory segment where the value is placed.
- It then stores the value from the physical address (local address*4) to a temporary variable and return the value.

➢ *getArrElem* - This function takes the name of the array for which the user wants the value and the offset (position of the element in the array) as parameter.

- It will then search the symbol table for the variable. If not found it will print a message that no such variable exists.
- It then extracts the logical address from the symbol table of the array element, using that we will find the local address from the page table of the memory segment where the value is placed.
- It then stores the value from the physical address (local address*4) to a temporary variable and return the value.

➢ *newFunc* - It will be called at the start of a function to indicate the starting of a function, it will insert a special character in the stack to identify the start of a function.

➢ *endFunc* - It will be called at the end of a function to indicate the ending of a function. It will clear the stack of all the local variables created in that function that is ending.

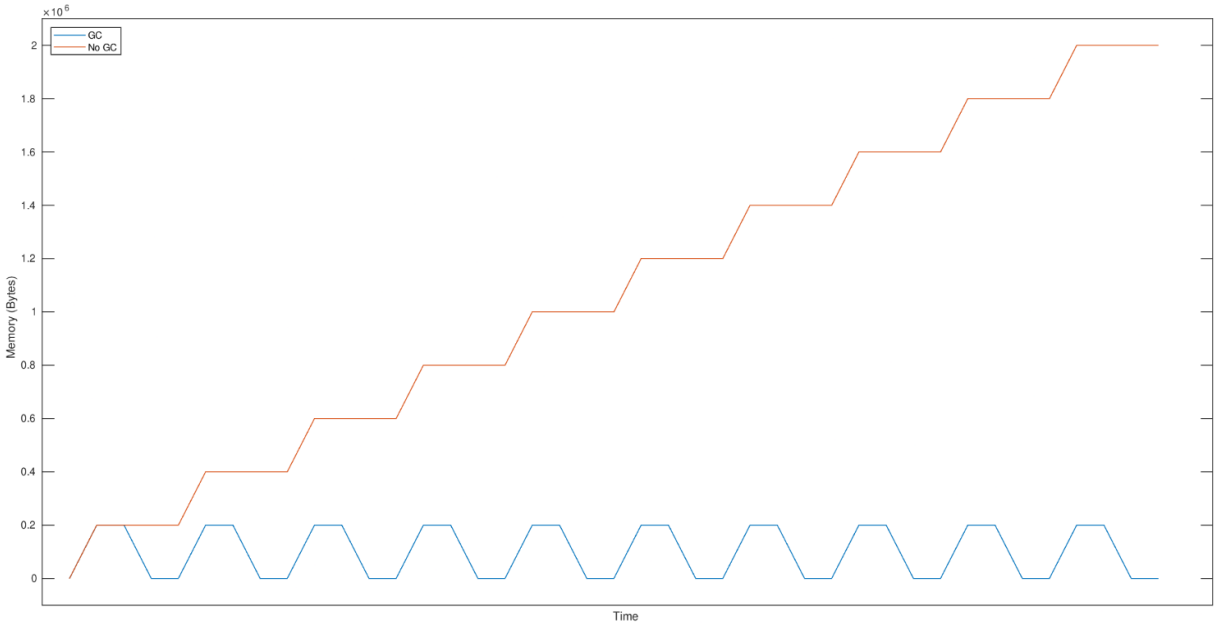
It will pop all elements of the stack until the special character added is encountered which indicates the end of function.

*For non-void functions, endFunc must be called by the caller function so that the return value is not discarded.

➢ *freeMem* - This function is called at the end of the program inside main, when all functionalities are over and we need to free the allocated memory. It will free all the dynamically allocated memory along with closing the garbage collector thread.

3) *What is the impact of mark and sweep garbage collection for demo1 and demo2. Report the memory footprint with and without Garbage collection. Report the time of Garbage collection to run.*

Demo1:



With Garbage Collector:

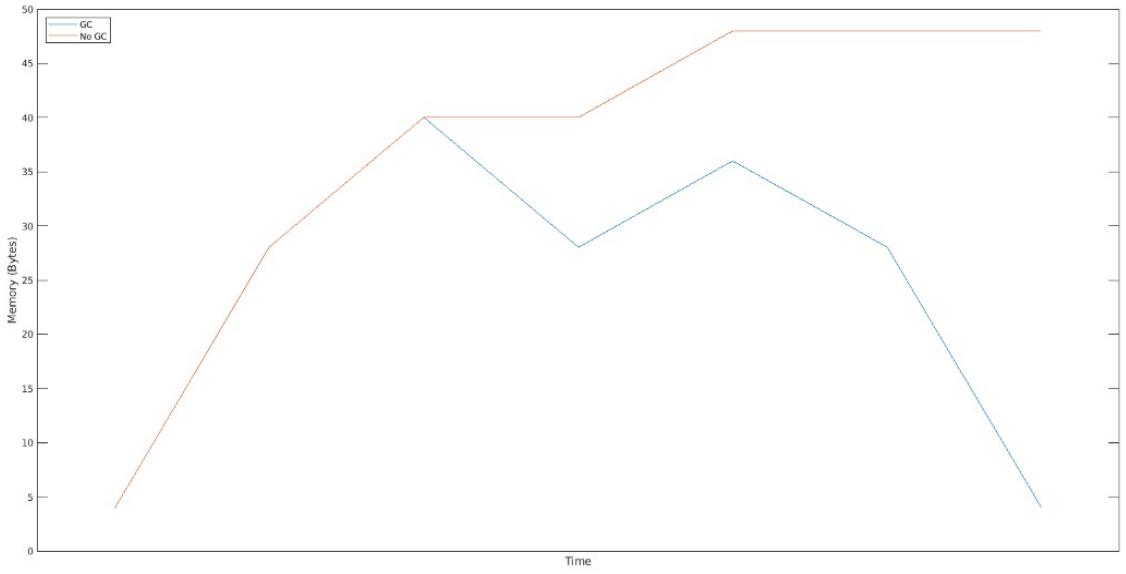
- ☐ Max: 200036 bytes
- ☐ Mean: 97594 bytes
- ☐ Time to run the code: 8.290668 s

Without Garbage Collector:

- ☐ Max: 2000072 bytes
- ☐ Mean: 1073223 bytes
- ☐ Time to run the code: 5.730602 s

Time to run Garbage Collector = 8.290668 - 5.730602 = 2.560066 s

Demo2: Using parameter k = 6



With Garbage Collector:

- ☐ Max: 40 bytes
- ☐ Mean: 24 bytes
- ☐ Time to run the code: 0.005302 s

Without Garbage Collector:

- ☐ Max: 48 bytes
- ☐ Mean: 36 bytes
- ☐ Time to run the code: 0.002821 s

Time to run Garbage Collector = 0.005302 - 0.002821 = 0.002481

From the above graphs we can see that without using the garbage collector, the memory footprint increases with each time stamp in a linear fashion.

But if we use a garbage collector in our library, we can see that the memory footprint decreases implying that memory is being freed.

4) *What is your logic for running compact in Garbage collection, why?*

The compaction algorithm goes as follows.

1. Whenever the use of a variable is over (explicitly or scope ended) the freeElem function is called with a variable or array name.
2. We then remove that variable or array from our memory using the sweepPhase function, which will create holes in our page table.

3. Periodically we are calling our garbage collector inside a thread which will do compaction. In compaction we are trying to fill those holes created, and aggregating all the holes at the end.
4. We will traverse the page table to find the first hole and if the hole is found we will mark it.
5. For any page (not hole) after the found position try to shift it as up in the page table as possible in order to fill all the holes.
6. At the end of our iteration we will find all the holes aggregated at the end.
7. We chose this compaction algorithm for our memory management because it takes linear time without using any extra space.

5) ***Did you use locks in your library? Why or why not?***

Yes, we have used locks in our library because our garbage collection is done by a separate thread.

It is done because the garbage collector effectively updates shared data structures of our memory management so to prevent that we have to use locks to stop data corruption.

********<u>*Demo files*</u>:

We have created 4 demo files:

a. demo1:
   As per the problem statements, 10 functions are created after taking 250 MB memory. Each function will create and populate an array of 50000 elements of the same data type with random data, destroy the array and return.

b. demo2:
   Implemented a fibonacci function in our code using memory management. Our main will take a parameter k using command line argument, and pass it to a function fibonacciProduct which will call the fibonacci to populate an array of first-k fibonacci numbers, compute its product and return the product.
c. demo3:
   Tested the character array features, wherein word alignment comes handy.
d. demoE:
   Tested the errors (type error, name error, memory error etc.)

********** END **********

Group Number = 34
Group Members :

Amartya Mandal	-	19CS10009
Divyansh Bhatia	-	19CS10027