OS Lab
Assignment-6 Report

Hardik Pravin Soni (**20CS30023**)

Yatharth Sameer (**20CS30059**)

Subhajyoti Halder (**20CS10064**)

Archit Mangrulkar (**20CS10086**)

- **What is the structure of your internal page table? Why?**
  Our page table has the following parts:
  - An array indexed by the local address that stores the physical address of the variable. This is the classic page Table necessary to access the memory locations of the Variables and read/modify their values as the user will give the logical address of the variables.
  - Stored the list of Variables that have been declared till now again indexed by their local address. This list contains the details of Variables, including their name, type, size, local address, array length (1 if it is not an array and the array size if it is), and a flag that marks the variables to be cleaned by the garbage collector. This is like the symbol table, which is necessary to know any information about the variable as and when necessary. For example, the type is needed for type checking, then the size is when we are freeing and allocating space to the variable, and so on.

  We have indexed by the local address so that we have an O(1) access variable.

1. **What are additional data structures/functions used in your library? Describe all with justifications.**
   Additional Data Structures:
   - **Stack**: We have implemented a stack to maintain the variables that are currently in scope. Variables are pushed into this stack when they get in scope and are popped from this stack when their scope ends. The Stack stores only pointers to the Variables to avoid extra memory usage
   - **Physical Address to Local Address Mapping**: We have implemented a mapping between the physical address and the local

address too. This is used when we are running compaction as we have the physical address and need to update it and save it on the page Table and variable list at the local address index.
- **Variable**: This is made to store all the details of the variable, that is, its name, type, size, local Address, array length (1 if it is not an array and the array size if it is), and a flag that marks the variables to be cleaned
- **map<char*, int> mp** - This map is used to store the mapping between the variable name and its local address. It helps in accessing the memory location of a variable using its name, which is useful for variable assignment, reading or writing.
- **Node, DLL** - These data structures are used to implement the linked list, which stores the value of a linked list variable. A linked list is a dynamic data structure that can be resized during runtime, so it is necessary to allocate and free the memory space for it dynamically.

Additional Functions:
- **isValid** - determines whether the given variable type and name are valid for use
- **getSizeFromType** - determines the size in bytes of a variable of a given type
- **CreateVariable** - creates a variable and initializes its properties
- **createStack** - creates a stack for storing variables
- **push** - pushes a variable onto the top of a stack
- **pop** - removes the top variable from a stack
- **top** - retrieves the top variable from a stack
- **isEmpty** - checks if a stack is empty
- **getSize** - returns the size of a stack
- **init** - initializes a character array
- **createMem** - creates a block of memory for storing variables
- **createVar** - creates a new variable in the memory block and updates the relevant data structures
- **max** - returns the maximum of two integers
- **typeCheck** - checks if a variable at a given address has a specified type
- **getTypeString** - returns a string representing the given variable type
- **assignVal** - assigns a value to a variable in the memory block based on its name and offset

2. **What is the impact of freeElem() for merge sort? Report the memory footprint and running time with and without freeElem (average over 100 runs).**

freeElem() is essential because it allows us to free the small-sized linked lists created during Merge Sort.

Without this, the **memory footprint would be persistently high** as the smaller-sized lists aren't cleared for usage, but it incurs **lesser time overhead** for clearing lists each time the scope ends.

NOTE: For a list of 50k elements, the number of leaf nodes = 50000, and the size of each element = 12 bytes.

Hence, the total size for leaf nodes only = 12 * 50000 = 600 KB!
(Actual memory usage is much higher as each element is allocated a single page, so actual memory footprint = 128 * 600 KB = **75 MB (~30% !)**)

For 100 iterations,

Avg Time: ***2.53125s*** **Max page usage:** ***52942 (31%)***

If freeElem() is called aer each list is finished being used, the **memory footprint decreases,** but **more time is taken** for each such list clearing.
For 100 iterations,

Avg Time: ***2.63414s*** **Max page usage:** ***391 (0.23%)***

3. In what type of code structure will this performance be maximized, and where will it be minimized? Why?

Such a memory management implementation would have **maximum performance** if the code structure **doesn't involve many scope definitions**, while the **performance would be hampered if the implementation involves multiple scope definitions** (typical of recursive functions). This is because more time overhead would be required for maintaining scopes using stack frames and then deleting local lists when the scopes end.

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

Yes, we have used locks in our code. We have used it to make sure that while compacting the memory, we don't create a variable/array or assign any value to the variable/array element. If a variable is being moved to a new location, we can either update the table first and then move the variable value or the variable value and update the table later. In both cases, assigning the variable a new value might assign the value to the wrong location or wrong time. Also, if we are moving the free spaces, then a wrong free space may be allocated to the new array or variable.