OS Assignment 6

Group Number 8

Mradul Agrawal: 20CS30034
Sourabh Soumyakanta Das: 20CS30051
Grace Sharma: 20CS30022
Umang Singla: 20CS10068

The internal page table is implemented in the form of a **map** named **pageTableEntries** which maps the name of the memory location to a stack of page table entries. The stack is necessary for bookkeeping purposes in case the memory segment is passed into another function. These entries are stored in the form of a struct named **struct pageTable** which has its fields : **physicalAddr** which is a pointer to the actual memory location, **numElements** which is the count of number of elements in the list. This is necessary because we need to have the actual location from where contents of the list start and the size of the list which can give us the memory locations that can be accessed without any memory access violations.

Additional data structures used:

- **map<string, stack<pageTable>> pageTableEntries** : This data structure stores a mapping of a memory segment name to the stack of pageTable which is necessary for memory access to the same segment even if the name changes during passing the memory segment to another function.
- **struct pageTable**
  - **physicalAddr** : This stores the actual memory address of the first block of the memory segment.
  - **numElements** : This stores the no. of units of that particular memory segment which can be used for access by index and preventing memory access violation.
- **struct block**
  - **name** : This stores the name of the assigned memory block.
  - **size** : This stores the size of the assigned memory block.
  - **free** : This stores whether the concerned memory block is free.
  - **pointer** : This stores the pointer to the starting address in the allocated memory segment.
- **list<block> blocks** : It is a list to store all the blocks present in the allocated memory segment.
- **void *memStart** : This refers to the actual address of the first unit of the large memory segment that was allocated during createMem.
- **int memSize** : This is the size of the memory segment that was allocated during createMem.

Additional functions used:

- **void printBlockList()** : It prints the details of all the blocks present in the allocated memory segment.
- **void printList(string name)** : It prints the details of all the units present in the block with the given name.

- **int getVal(string name, int offset)** : It returns the contents present in the block with the given name at the given offset.
- **void startScope()**, **void endScope()** : Used for stack modification purposes.
- **int memoryFootprint()** : It calculates the amount of memory which is not freed.

No locks were used in our library since our code implementation is single threaded.

Execution time and Memory Footprint analysis:
Following is the statistics after a mergesort function call have been completed:
With freeElem = true

1. Average running time: 1.1 seconds
2. Average memory footprint: 0 bytes

With freeElem = false

1. Average running time: 39.33 seconds
2. Average memory footprint: 20027136 bytes

Note that above analysis is done after the merge sort has been executed completely and thus we can observe that memory footprint for the freeElem = true case is 0, that means after the code have been completely executed, the memory is completely cleaned and to the contrary a lot of memory (20027136 bytes) is unused after the code is completely executed and is garbage to the user.

Clearly, a lot of time is spent on an average for the case in which code is executed with freeElem = true is set, as the good malloc program searches through the allocated memory if there is a free block to allocate the list. Now, as the memory is not freed when freeElem is not used, the program has to search from a long list of allocated blocks to fetch a free block and thus a lot of time is spent every time when a list allocation is needed, thus explaining the time difference.

Regarding the code structure and implementation through which performance could be maximised.
The current implementation in which a list of block allocation bookkeeping and a map of string and stack is used to keep the information about the currently allocated blocks is kept is very fast and memory efficient. Map enables the searching for the physical address to the name of the list very fast using hashing techniques and a doubly linked list of allocated and free blocks enables the library to search for the free blocks by iterating over it using First Fit Strategy.

We also provide startScope() and endScope() functionality for the user, it is up to the user which method to use for allocating the memory. The above functions could be used at the start and end of a function call and it will automatically clear the allocated memory according to the global stack which is maintained by the library itself. The above results are reported without these scope() functions.