Assignment 5 Report

Virtual memory

Group 54

Aryan Singh, Abhinandan De

March 31, 2022

CS39006

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

The page table has been designed in a hierarchical fashion.

1. Firstly, we have an *enum* representing all the available types.
2. Then we have a generic *struct Object* for all types of variables in our program.
   - It has the following attributes:
     - objType: Indicates the type of the object.
     - size: Indicates the size of an individual element for an array or total size for a single variable.
     - totSize: Total size occupied by the variable in memory (after padding).
     - symTabIdx: The position of the object in the symbol table.
   - It has the following methods:
     - constructor/destructor: Default behaviour.
     - ostream& operator<<: Used for printing the object.
3. After that we have a *struct symTabEntry* representing a symbol table entry.
   - It has the following attributes
     - lock: Used for implementing locking behaviour.
     - wordIndex: Index of the variable in the memory created using createMem.
     - valid: Indicates whether it points to a valid memory location.
     - marked: Indicates whether an entry in the symbol table is still alive.
   - It has the following methods:
     - Constructor/destructor: Default behaviour.
     - unmark: Used to mark a variable as dead (done by garbage collector)
     - invalidate: Used indicate that memory is invalidated.
4. Lastly we have our symbol Table *struct entries.*
   - It has the following attributes:
     - validMem: A bitset that stores if a word in memory is valid or not.
     - myEntries[ ]: An array of symbol Table elements.

  - listOfFreeIndices: A stack containing the free indices in the symbol table.
  - validMemlock: Implementing locking behaviour while locking.
- It has the following methods
  - constructor/destructor: Default behaviour.
  - insert: Inserts an entry into the symbol table.

We adopted a hierarchical design to make things clear and sacrosanct. For each variable, we create two things:

- An instance of the object.
- Its corresponding entry.

After that we find a fit in memory using *getNextFit* and update the word index in the symbol table. Initially it is marked(alive) and is valid(if the allocation is successful). We also update the symbol table index for the corresponding object to facilitate faster access.

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

Apart from the data structures mentioned above, we used two additional data structures

1. A stack: We create an instance for storing variables in scope and another one for storing free indices in symbol table.
   - It has the following attributes:
     - indices[ ]: An array that mimics the stack.
     - top: Variable that points to the top of the stack.
     - lock: Ensures locking behaviour.
   - It has the following methods:
     - constructor/destructor: Default behaviour.
     - push: Pushes an element into the stack.
     - pop: Pops an element from the stack.
     - peek: Returns the top element of the stack without popping.

2. A bitset: This bitmap checks if a word in memory contains valid data or not.
    - It has the following attributes:
        - ptr: Pointer to the current head of the physical memory.
        - sizeAvl: Size available if we allocate memory starting from ptr
        - totSizeAvl: Total size available in the memory.
        - mem[ ]: The actual bitset (array) indicating if a word is valid.
    - It has the following methods
        - constructor/destructor: Default behaviour.
        - getIndex: Returns the index in the mem array for a given word.
        - getOffset: Returns the offset in the mem array for a given index.
        - isSet: Returns if a word in memory is valid.
        - set: Sets the corresponding bit in the memory thus validating a word in memory.
        - reset: Resets the corresponding bit in the memory thus invalidating the word in memory.

The functions that we used are:

1. createMem: Creates the memory segment of memSize bytes
2. createVar: Creates a variable of a given type
3. assignVar(*overloaded*): Handles assignments involving non-array elements.
4. getVar: Returns the value of a variable from memory.
5. createArr: Creates an array of a given type and length.
6. assignArr(*overloaded*): Handles assignments involving array elements.
7. getArr: Returns the value of an array element from memory.
8. getSize: Returns the total size required for a variable.
9. getNexttFit: Gets the next fit in memory i.e. essentially the one pointed by *ptr*. First a check is done on the total available memory, if not sufficient, we throw an error. If it doesn't fit, a compaction is performed first and then the object is stored.

We use the *getNextFit* algorithm with compaction to obtain a better time complexity. Our algorithm finds the memory location for storage in an amortized $\mathcal{O}(1)$ complexity. This time complexity is better than that of both *bestFit* and *firstFit* algorithm in terms of time complexity ($\mathcal{O}(n)$ on average with n being the no. of words). Although since we maintain a bitmap for tracking if a word in memory, and thus incur a significant overhead in terms of space complexity($\mathcal{O}(n)$), we argue that the worst case complexity is same for both *bestFit* and *firstFit* algorithms. We also optimize our valid memory indicator using a bitmap which uses 1 unsigned integer with 32 bits to indicate the validity of 32 contiguous words in memory.

10. freeElem: Frees the element from memory by marking its entry in symbol table as invalid.
11. mark: It unmarks all the variables that are dead when they go out of scope. This is done using a stack that keeps track of variables in scope.
12. sweep: This just checks if a variable is unmarked and still has a valid memory. If it does, it calls *freeElem*.
13. compact: This compacts the memory using a two pointer approach.
14. gcRun: This sequentially calls *mark*, *sweep* and *compact*. The mark and compact functions are called optionally if specified in the function parameters.
15. gcRoutine: The other thread routine which periodically calls *gcRun*
16. copyWordWise/copyWordWiseGet: These functions are for the cases when we have to read/write an array element and it is spread across two words.

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.

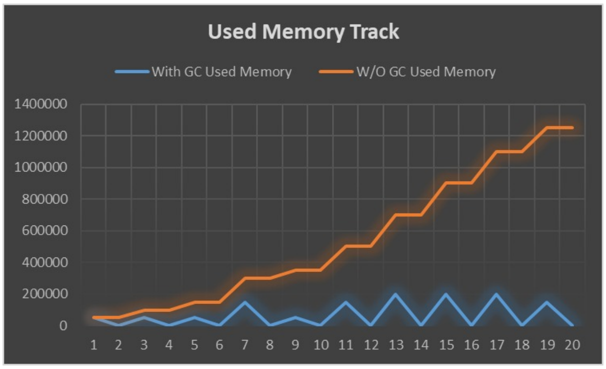
Mark and sweep garbage collector is vary useful in reducing the memory footprint of the functions. The *gcRun* call at the end of a scope reduces the memory footprint. It essentially becomes similar to calling one function instead of 10 while running *demo1.cpp*. However running the garbage collector adds some overhead. We observed an additional runtime of 58.32 ms for a full run. If we remove the compact operation, we observe an avarage runtime of 29.12 ms in case of *demo1.cpp* where we allocate 256 MB memory using *createMem*.

We have contrasted two variables here

1. Total used memory: Total valid memory in the memory block.

With GC	Without GC
50036	50036
32	50036
50036	100040
32	100040
50036	150044
32	150044
150036	300048
32	300048
50036	350052
32	350052
150036	500056
32	500056
200036	700060
32	700060
200036	900064
32	900064
200036	1100068
32	1100068
150036	1250072
32	1250072

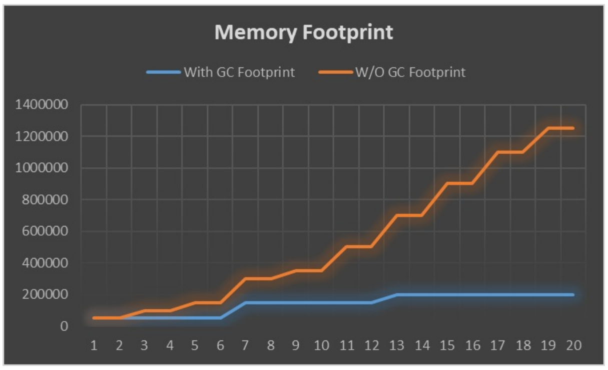
Table 1: Variation of used memory (in bytes)



2. Memory footprint: Memory that has been *touched* by our library

With GC	Without GC
50036	50036
50036	50036
50036	100040
50036	100040
50036	150044
50036	150044
150036	300048
150036	300048
150036	350052
150036	350052
150036	500056
150036	500056
200036	700060
200036	700060
200036	900064
200036	900064
200036	1100068
200036	1100068
200036	1250072
200036	1250072

Table 2: Variation of memory footprint (in bytes)



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

We have used a simple *2-pointer* approach in compaction. The left ptr keeps track of the next free word and the right ptr keeps track of the next word to be copied. This essentially compacts all the memory to the left side and removes holes present in between. The time complexity of this approach is $\mathcal{O}(n)$ where n is the number of words in memory.

Although it reduces memory footprint, *compact* is a very expensive operation because it runs in linear time w.r.t. the no. of words. Thus we encounter the very famous *time-memory tradeoff* while running compact.

Compact may be invoked in 3 occasions

1. End of scope: We always compact after the end of scope.
2. Failure to allocate a variable in *createVar*: We are compelled to compact in this case.
3. By the other thread: We call compact once every five times. Rest of the times, it just performs the *sweep* operation. We never call the *mark* function since we may/may not be at the end of a scope.

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

We had to use locks since there are 2 threads simultaneously running in our library. Since they may access resources in a parallel fashion we might observe some corruption in our data.

We used the following locks in our library:

1. For each symbol table entry: This prevents the other thread from invalidating the memory in use.
2. For accessing the memory: We used one lock for the whole memory segment. Else it would incur a lot of overhead if we used one lock per word.
3. For accessing the scope stack.
4. For preventing *compact* and *getNextFit* from running together.