Operating Systems Laboratory (CS39002)

Assignment-5 Report

Suhas Jain - 19CS30048

Parth Jindal - 19CS30033

Contents

1 Internal Symbol table Structure

The internal symbol table stores information about the location in memory (word-id and offset in that word) of all the primitives and arrays which have been currently allocated.

```
struct Ptr {
    Type type;
    int addr;
    Ptr(const Type& _t, int _addr) : type(_t), addr(_addr) {}
};

struct ArrPtr : public Ptr {
    int width;
    ArrPtr(const Type& t, int _addr, int _width) : Ptr(t, _addr), width(_width) {}
};
```

`Ptr` is an abstract class mirroring the C pointer, it stores the local address as well as the type of the object. `ArrPtr` is a derived class which along with the local address and type that `Ptr` class stores, also stores the width of the array. The local addresses (`int addr`) stored here is nothing but 4 times the index of that particular primitive or array in the internal symbol table. Declaring the `Ptr` and `ArrPtr` help us store the local addresses directly with the pointer which can be accesses in constant time, rather than doing string matching over a list of declared variables or using a hash-map approach, both of which can be linear in worst case.

```
struct Symbol {
    unsigned int word1, word2;
};

struct SymbolTable {
    unsigned int head, tail;
    Symbol symbols[MAX_SYMBOLS];
    int size;
    pthread_mutex_t mutex;
    SymbolTable();
    ~SymbolTable();
    int alloc(unsigned int wordidx, unsigned int offset);
    void free(unsigned int idx);
    inline int getWordIdx(unsigned int idx) { return symbols[idx].word1 >> 1; }
    inline int getOffset(unsigned int idx) { return symbols[idx].word2 >> 1; }
    inline void setMarked(unsigned int idx) { symbols[idx].word2 |= 1; }
    inline void setUnmarked(unsigned int idx) { symbols[idx].word2 &= -2; }
    inline void setAllocated(unsigned int idx) { symbols[idx].word1 |= 1; }
    inline void setUnallocated(unsigned int idx) { symbols[idx].word1 &= -2; }
    inline bool isMarked(unsigned int idx) { return symbols[idx].word2 & 1; }
    inline bool isAllocated(unsigned int idx) { return symbols[idx].word1 & 1; }
    int* getPtr(unsigned int idx);
};
```

These are the struct declarations we make for a symbol entry (`struct Symbol`) and the internal symbol table (`struct SymbolTable`). We discuss the use of each data field in these structures along with algorithm and functionality of each method specified.

1.1 Symbol Table Initialisation

The size of symbol table is initialized at runtime depending upon the amount of memory specified in $createMem()$. The Stack is also of the same size as that of the Symbol Table.

$$\textbf{sizeof\_symbol\_table} = \textbf{min}(\mathbf{2^{15}}, \lceil \frac{\textbf{sizeof\_memory}}{\mathbf{12}} \rceil$$

where 12 is sizeof smallest atomic object(8 for header, footer and 4 for object.).

1.2 Free List in Symbol Table Entries

Whenever we want to make a new entry in the table we have to find an index which is unallocated. Iterating through the table and finding the first unallocated entry is linear and hence can be expensive. We propose a linked list type constant time approach. When a particular entry is unallocated, we do not need to store any offset and hence we can use these 31 bits in `word2` data field (read about `word2` to understand better) to store the index of next available free entry in the symbol table. We also have 2 fields `unsigned int head` and `unsigned int tail` to store start and end of the free list. Whenever we need to create a new entry we make it at `head` position and shift the head, and whenever we need free a old entry we append it after `tail` position and shift the tail. Thus all memory access, creation and deletion of a symbol in symbol table become $\mathbf{O(1)}$

1.3 Data Members in `struct Symbol`

- **unsigned int word1**: Least significant bit of `word1` stores whether the particular Symbol table entry is allocated or not. Rest of the 31 bits store the word-id of the symbol, i.e. the word offset from the start of the allocated memory.
- **unsigned int word2**: Least significant bit of `word2` stores whether a particular Symbol table entry is marked or not, by marked we mean marked in context of mark and sweep algorithm. If a particular entry is not marked and currently allocated it will be unallocated and the corresponding memory block will be freed next time garbage collector executes. Rest of the 31 bits store the offset in bytes in a particular word.

1.4 Data Members in `struct SymbolTable`

- **unsigned int head**: Stores the index of the first symbol table entry in the free list of the internal symbol table.
- **unsigned int tail**: Stores the index of the last symbol table entry in the free list of the internal symbol table.
- **Symbol* symbols**: List of declared primitives and arrays in the symbol table. Each entry in this array stores flags, word-id and offset as explained in the section above.

- **pthread_mutex_t mutex**: Lock to allow mutual exclusion to prevent data corruption and deadlocks.

1.5 Member Functions in `struct SymbolTable`

- **SymbolTable()**: Constructor of the `SymbolTable` object. It initialises head, tail and stores value of index in most significant 31 bits of `word2` of all symbol table entries as initially whole table is unallocated. Also initialises the mutex lock of the table.
- **~SymbolTable()**: Destructor of the `SymbolTable` object.
- **int alloc(unsigned int wordidx, unsigned int offset)**: Create a new entry in the symbol table. As we maintain a free list, it is made at `head` position in the table. Returns the value of index which is allocated, if the table is full it returns -1.
- **void free(unsigned int idx)**: Unallocate an already allocated block in the symbol table and update the `tail`.
- **int* getPtr(unsigned int idx)**: Return the pointer in the actual memory for the entry at the index `idx`.

1.6 Variable Pointer to Memory Address Translation

To access or update the value of a variable (might be a primitive or array element) we need to find the memory address of that variable, we get it through the following schemantics:

- As mentioned earlier each object of type `Ptr` has a `addr` data field which is a multiple of 4, this serves as our local address.
- This local address divided by 4 is nothing but the index of this variable in the page table.
- 31 bits of `word1` field for each page table entry store the offset (`word-id`) in words from the start address of the allocated memory segment through initial `malloc`. Thus, this address can be added to the start address to get the address of the variable.
- For arrays where multiple elements can exist in the same word (`bool` or `char` array), we might also need to add an additional offset within a particular word.

2 Additional Data Structures / Functions Used

2.1 Memory Segment

While running a program whenever a new variable or array is needed we need to find free spaces in memory which can accommodate the new variable or array. Thus we need size and other information like is the block free or used about a particular memory block. This information is also useful when we do compaction or coalesce two contiguous free memory blocks. For this, we implement a implicit free list where we store this information about the blocks in the initially allocated main memory segment itself without need of any other external data structure with separate memory.

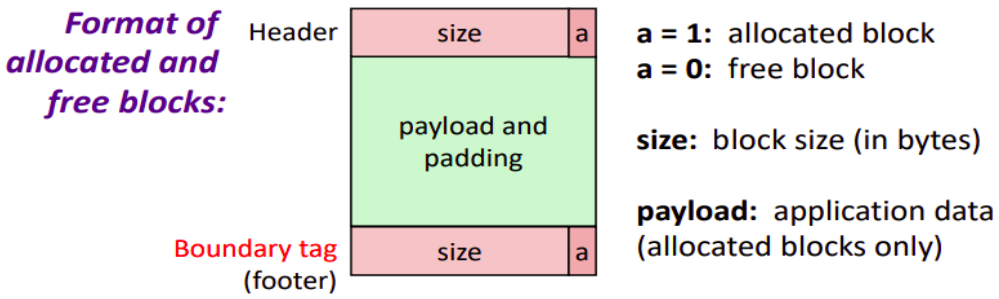

Figure 1: Structure of a Memory Block, Image courtesy: CSE351, Washington University

We define a word as a segment of 4 bytes and our whole memory management works on word level rather than byte level. A block of memory might have 1 or more words and is either free or used to store a single variable or array. For each block of memory we have a header of size 1 word and a footer of the same size. When a block of memory starts the first word is the header. The fist 31 bits of the header store the size of the block in words (with header and footer). The last least significant bit stores weather the block is free (0) or allocated (1). Note that footer stores the exact same information as the header.

2.1.1 Free List Traversal

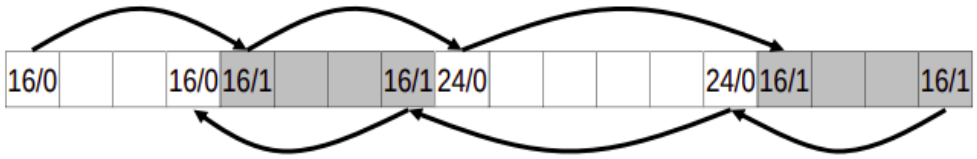

Figure 2: Traversing Free List Using Header and Footer, Image courtesy: CSE351, Washington University

Using header we can traverse the memory segments in blocks, when we read the header we get the size of the block so we can directly jump to the memory address of the next block without reading all the contained words. When we are at an address `p` in the memory (which is an address of an header), we can compute the address of the next block by computing `(p + (*p >> 1))`. We can also check if the block is free or allocated by checking the value of `(*p & 1)`. We might need to traverse the memory blocks sequentially from the reverse direction while coalescing blocks with its next and previous blocks after freeing hence we maintain the same information in both header and footer.

2.1.2 Finding Free Block using Free List

We traverse through the free list from the start and keep on advancing our current pointer to the header of the next memory block. When we read a header we check weather the memory block is free or not and we also check if the memory block is of sufficient size or not. Basically, we follow a **first-fit** strategy of allocating the memory. This approach takes $O(n)$ time, n being the number of blocks in memory. But since we are using a free list n is usually much less than total number of words in memory hence this is efficient.

2.1.3 Allocating a New Block using Free List

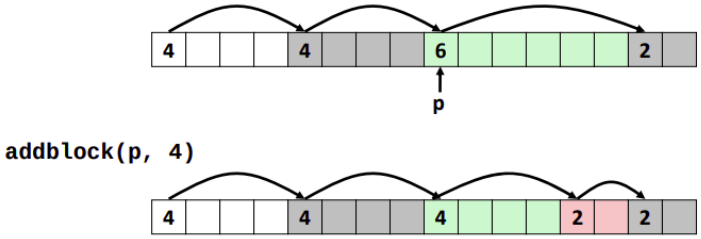

Figure 3: Splitting a block into an allocated and a free block, Image courtesy: CMU, Qatar

After we have the block where we can store our variable or array we need to change it's structure since the allocated space might be smaller than the free space, we might have to split the block into an allocated block and a free block. This requires creating a new header and a footer and changing previous ones, which can be done in $O(1)$ time.

2.1.4 Freeing a Block using Free List

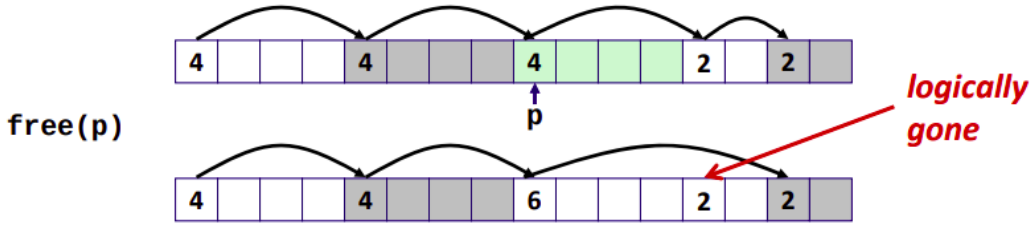

Figure 4: Traversing Free List Using Header and Footer, Image courtesy: CMU, Qatar

When a block is freed from memory the allocation bit is modified in both its header and footer to denote that it is a free block. Apart from this we also need to check if the next and/or the previous block is free or not. If it is free then we need to combine the adjacent free blocks to maintain our free list property. This involves changing a few headers and footers and can be done in $O(1)$ time

2.1.5 Why use a free list?

Maintaining a free list makes sense from point of view of time complexity, memory overhead as well as simplicity of implementation. As all the information in the free list is stored in the allocated memory itself we do not require any external data structure to keep rack of the free and used blocks, and hence it is very simple and clean to implement. We save space as we do not have to store memory addresses of each block which we will have to store if we use any external data structure. Except finding a first-fit memory block all the operations are $O(1)$ and require only a couple of steps. We coalesce blocks whenever we can and regularly do compaction to keep the size of blocks maximum and number of blocks minimum. As mentioned earlier finding a free block is also much more efficient than it appears to be as number of blocks usually much less than total number of words in memory.

2.2 Global Stack

```
struct Stack {
    int _top;
    int* _elems;
    int capacity;
    Stack(int size);
    ~Stack();
    void push(int elem);
    int pop();
    int top();
};
```

For maintaining scope information and garbage collection we need to keep track of variables in order of their declaration. We use a global stack where we push special characters to denote start of scope (read more in garbage collection) and then push the variables as they are declared. This helps us in mark phase of the mark and sweep algorithm of garbage collection. As shown above, we declare a basic stack with basic stack functionalities and allocate some amount of memory initially (check section 1.1 for more info) when we call the `createMem` function.

2.3 Library Interfaces

We develop a library using which we can declare variables and arrays of 4 types. We can read and write values in them and free them when required. To do all this we provide some library interfaces which are as follows:
Note: `__type__` is used as a placeholder here for type of variable in case of function overloading.

- `void createMem(int size, bool gc = true)`: Used to initially allocate memory for all purposes as well as initialise the garbage collector thread.
- `Ptr createVar(const Type& t)`: Create new primitive of the mentioned type. Used as follows:

```
Ptr a = createVar(Type::INT);
Ptr b = createVar(Type::MEDIUM_INT);
Ptr c = createVar(Type::CHAR);
Ptr d = createVar(Type::BOOL);
```

- `void getVar(const Ptr& p, void* val)`: Get value stores in a variable into another locally declared object which is passed by reference. Used as follows:

```
Ptr k = createVar(Type::INT);
// Assign some value to k
int val;
getVar(k, &val);
// Value of k will be copied in val
```

- `void assignVar(const Ptr& p, __type__ val)`: Function used to assign a given value to a declared variable. Function overloading is used to make it compatible with all the 4 types. Used as follows:

```
Ptr a = createVar(Type::INT);
assignVar(a, 10);
Ptr b = createVar(Type::MEDIUM_INT);
assignVar(b, medium_int(30));
Ptr c = createVar(Type::CHAR);
assignVar(c, 'a');
Ptr d = createVar(Type::BOOL);
assignVar(d, true);
```

- `ArrPtr createArr(const Type t, int width)`: Creates a new array of the given `width` and type. Used as follows:

```
ArrPtr arr1 = createArr(Type::INT, 50000);
ArrPtr arr2 = createArr(Type::MEDIUM_INT, 50000);
ArrPtr arr3 = createArr(Type::CHAR, 50000);
ArrPtr arr4 = createArr(Type::BOOL, 50000);
```

- `void assignArr(const ArrPtr& p, int idx, __type__ val)`: This function is used to assign the given value (`val`) to a element at particular index of the array. This function has total 8 overloadings, 4 of them are of this type, the other 4 are talked about in next point. It is used as follows:

```
ArrPtr arr1 = createArr(Type::INT, 50000);
assignArr(arr1, 100, 10);
ArrPtr arr2 = createArr(Type::MEDIUM_INT, 50000);
assignArr(arr2, 100, medium_int(10));
ArrPtr arr3 = createArr(Type::CHAR, 50000);
assignArr(arr3, 100, 'a');
ArrPtr arr4 = createArr(Type::BOOL, 50000);
assignArr(arr4, 100, false);
```

- `void assignArr(const ArrPtr& p, __type__ arr[], int n)`: Sets value of all the elements of an array (of size n) as the value of elements in a local array `arr` corresponding to the same indices. Used as follows:

```
ArrPtr arr = createArr(Type::INT, 10);
int arr_local[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
assignArr(arr, arr_local, 10);
// Similarly for all other types
```

- `void getVar(const ArrPtr& p, int idx, void* _mem)`: Get value stores at a particular index in an array. Used as follows:

```
// Assume ArrPtr arr contains {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int val;
getVar(arr, 5, &val)
// val will now contain the value 5 (element on index 5 in the array arr)
```

- `void gcActivate()`: Explicitly wakes up the garbage collection thread by sending a signal so that garbage collector starts running.
- `void initScope()`: Initialise a scope by pushing a special integer (-1 in our case) into the stack.
- `void endScope()`: End a scope by popping from the stack till we get the special integer (-1 in our case). Also unmarks the elements which are popped in the page table.
- `void freeElem(const Ptr& p)`: Frees the element which is passed. Unmarks the element in the page table then garbage collector removed the entry and from memory.
- `void freeMem()`: Deletes all the allocated memory, semaphores and mutex locks. Sets all the global pointers to `NULL` and terminates the garbage collection thread.

3 Garbage Collection Mechanism

Garbage collector thread is initialised after we allocate memory in `createMem` function by calling `garbageCollector` in the new thread. Garbage collector thread now performs some tasks which involve periodic garbage collection, compaction and garbage collection when signal is received. The mechanism of how all of this is performed using our version of the popular mark and sweep algorithm is as follows:

- We perform an efficient version of the mark step where we save repeated traversals through the stack. Whenever a functions begins execution, `initScope` is called and it pushes -1 into the stack as a unique character to signify the start of the stack. Now as variables are created in this scope its local address is pushed into the stack and mark bit is set to 1. When the scope ends (we call `endScope` explicitly), we pop from the stack till we reach -1 and set the mark bit of the popped variables as 0. Thus saving the overhead of doing multiple mark, unmarks for active variables.
- Now garbage collection thread has a while loop running till the thread terminates. In that thread it waits for some specified amount of time (stored in `GC_PERIOD_US` in our case, which is 10 microseconds) and then it calls `gc_run`. `gc_run` scans the entire symbol table and frees the symbol table entries which have their mark bits as 0 and valid bit as 1, which is essentially the sweep step in mark and sweep algorithm. The allocated memory is freed along with it's entry in the symbol table.

- Other than the periodic garbage collection, We perform garbage collection on demand. For doing this we provide `gcActivate` interface in our library which sends a signal to the garbage collection thread to run the garbage collection algorithm. In our demo files and as a convention we call `gcActivate` after the scope ends.
- Garbage collection thread also periodically checks a metric to find the extent of memory fragmentation (talked about in the next segment) and if it is high it calls the `compactMem` function which performs the memory compaction.

4 Compaction Logic in Garbage Collection

When we free and allocate memory dynamically, holes are created in our memory, in compaction first we detect when to do compaction with the help of some heuristic (mentioned below) and then combine all the holes (free spaces in memory) into one big free memory block so that bigger objects can be allocated. In this processes along with the memory symbol table also needs to be updated as word-id of its entries are changed.

4.1 When to do Compaction?

- Basically we have to detect when fragmentation is high and check the extent of fragmentation whenever our garbage collector runs and compact if it is above a threshold value.
- To check the extent of fragmentation we maintain a few data fields in memory struct, their store the following info:
  - `int totalFreeMem`: Total amount of free memory.
  - `int totalFreeBlocks`: Total number of free blocks in memory.
  - `int biggestFreeBlockSize`: Size of biggest free block in memory.

  Note that value of `totalFreeMem` and `biggestFreeBlockSize` is stored in terms of number of words. Also note that `biggestFreeBlockSize` is an estimation and not the exact value (read further for more details).
- We have to figure out how do we calculate the value of `biggestFreeBlockSize`. Calculating the exact value is not practical as it will require a iteration over all memory blocks, before compaction the number of holes is already large so doing this does not make sense. We devise a heuristic to calculate an estimate. Initially whole memory is free so we start with size of whole memory as the size of biggest free block. Now whenever we free some segment of our memory and a new block becomes free which is bigger than the current size of biggest block, we update the value. We also update the value when some chunk out of this biggest free segment gets allocated. Please note that size of biggest free block can never be less than the average size of a free block so we store the average in `biggestFreeBlockSize` if in some cases it becomes less than the average.
- As a metric we define a `free_ratio`, and whenever this ratio exceeds a particular value (2.5 in our case), we perform compaction. This ratio is defined as:

$$\textbf{free\_ratio} = \frac{\textbf{totalFreeMem}}{\textbf{biggestFreeBlockSize}}$$

- The biggest problem of fragmentation is that even if there is sufficient memory left, we might not be allocate objects of certain size as the free memory is present in different holes. So, intuitively we can see that how much a memory is fragmented can be characterised by the size of the biggest free block currently present. Before compaction size of biggest free block is stored in `biggestFreeBlockSize` and after compaction as all the holes are combined into one single free memory block, size of the biggest free block will be `totalFreeMem`. Intuitively using this ratio makes sense as this tells us how much bigger objects we will be able to allocate in our memory after compaction which we could not earlier.
  When a new memory block is freed:

  $$\textbf{biggestFreeBlockSize} = \textbf{max}(\textbf{avgSize},\ \textbf{new free block size},\ \textbf{biggestFreeBlockSize})$$

  When memory is allocated from biggest block:

  $$\textbf{biggestFreeBlockSize} = \textbf{max}(\textbf{avgSize},\ \textbf{biggestFreeBlockSize} - \textbf{allocated memory size})$$

4.2 How to do Compaction?

We use a compaction algorithm inspired by the `LISP2` algorithm which uses three different passes over the memory and its is completed in $O(n)$ time, n being the number of memory blocks (both allocated and holes) in memory. The three key steps are as follows:

- As after compaction the word-ids of the objects in symbol table will be changed we will have to preempt the new word-id of the objects in the symbol table after compaction, update the symbol table and then move on to compacting the memory. To preempt the word-id, if the total free space encountered till now is x words, the word-id of the present block will be x. We store this new offset in the footer word of that particular block itself, to avoid any space overhead, as the footer as no purpose during compaction this can be done without loss of information.

- In the second pass we update the symbol table entries with these preempted values. We iterate through the symbol table, go to the memory location of each of the entry, read value of new word-id from the footer of the block and update that particular entry in the symbol table.

- In the final pass we do the actual memory compaction and move memory blocks to their new locations. We iterate over the memory blocks and if the current memory block is free and next memory block is allocated we swap these two blocks. After swapping we also check if the next block if free, and if it is free we merge the free blocks. Please note that as we merge consecutive free blocks when we free some memory there can be no two free consecutive memory blocks during compaction. In this manner, we keep on iterating till the end till all the holes are removed and we have one big free memory block at the end of our memory.

5 Impact of Garbage Collector on Running Time

When we talk about the impact of garbage collector on running time of the code, the table below provides the average running times across 5 runs:

	With Garbage Collector	Without Garbage Collector
demo1.c	0.065s	0.054s
demo2.c	0.008s	0.007s

5.1 Justification

We can observe from the table that running time increases with garbage collector for both the demo files. As we will see next that a garbage collector helps a lot in reducing the memory impact of the code so it obviously has some amount of trade-off with the running time. Although garbage collector and main program run on separate threads, true parallelism is not achieved. Resources are shared between the threads and there's extra time consumed during switching between threads. One of the most important reasons for time to increase is that when we use mutex locks (read on further to know details on locking) sometime one thread has to wait for a process to end in the other thread.

6 Impact of Garbage Collector on Memory Usage

With garbage collector we expect a good amount of reduction in memory usage. Below are the graphs of memory usage with respect to instruction flow of demo1 and demo2. Please note that the blue line denotes the memory usage without garbage collection thread running in parallel and red line denotes memory usage with garbage collection thread running in parallel.

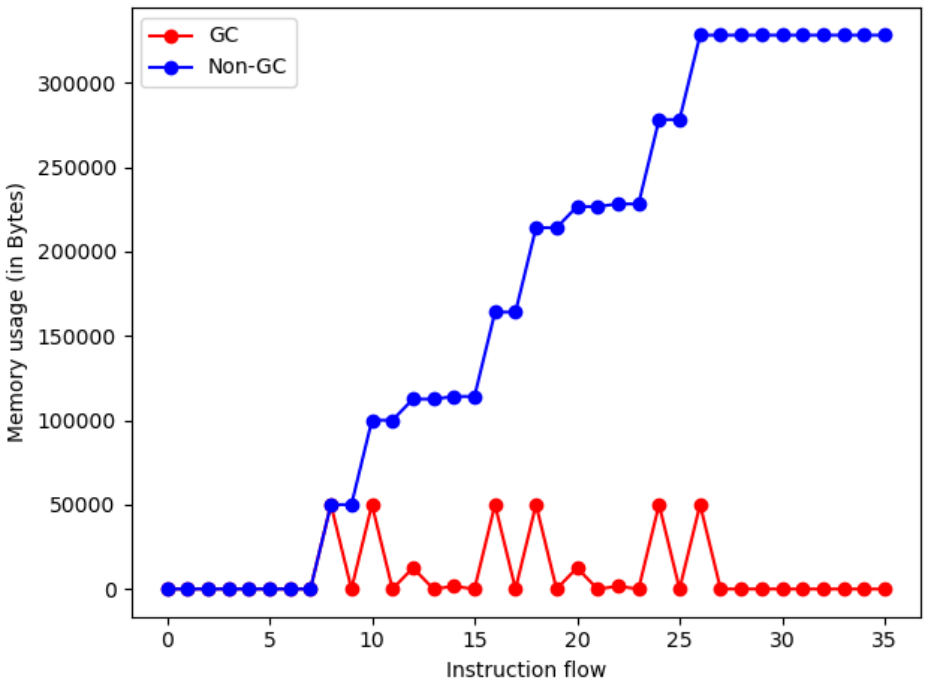

Figure 5: Memory Usage Plot of demo1.cc

	With Garbage Collector	Without Garbage Collector
Mean	9233.8B	173817.4B
Standard Deviation	18507.5B	126137.6B
Max	50026B	328170B

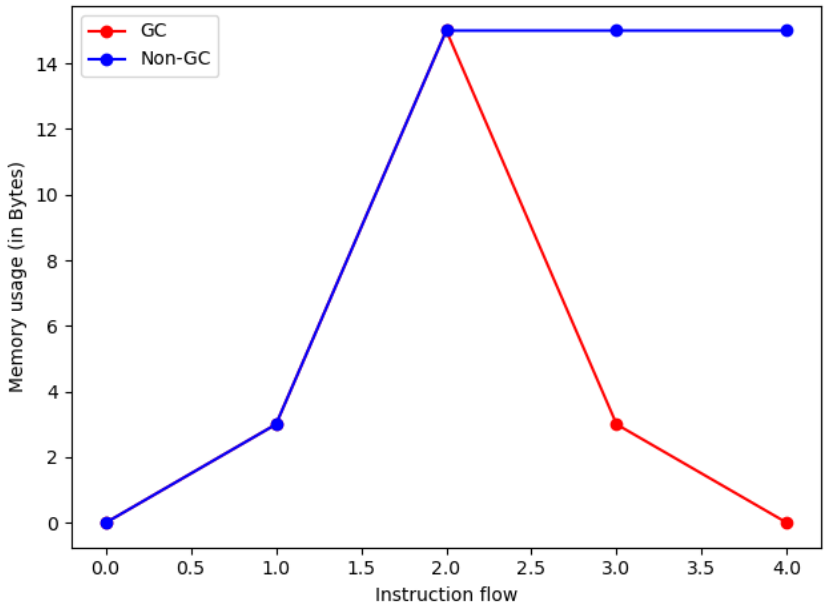

Figure 6: Memory Usage Plot of demo2.cc

	With Garbage Collector	Without Garbage Collector
Mean	4.2B	9.6B
Standard Deviation	5.56B	6.68B
Max	15B	15B

6.1 Justification

As we saw in sections before garbage collector checks for unmarked entries in the symbol table and removes them from memory, this process is also performed by explicitly starting the garbage collection thread by sending a signal. All this is done along with compaction which causes more efficient allocation of memory. Garbage collection ensures all the out of scope objects and explicitly unallocated objects are removed from memory as well as the symbol table. This phenomenon can be clearly noticed in the graphs above where we can see sharp decline in memory usage at regular intervals when garbage collector is running. Impact is seen in a much more magnified manner in `demo1.cc` as it has more number of operations and hence more scope for garbage collection.

7 Use of Locks

Two mutex locks have been used in our library. First one associated with with the memory segment and the second one associated with the symbol table. Need and functioning of both of these locks is as mentioned below:

- We have two threads operating at the same time, the main thread and the garbage collection thread. During compaction the garbage collector thread changes the position of blocks in memory, during normal garbage collection also status of blocks is changed from allocated to unallocated, along with that coalescing of blocks is also performed if necessary. If the main thread tries to access or create a variable from the main thread when the operations of garbage collector or compaction is underway, there might be data corruption or data loss which is undesirable.
- As we saw earlier in the compaction algorithm first addresses are updated for the entries in the page and then the locations of blocks are changed in the memory. So, if our main thread accesses the symbol table after it has been updated but memory hasn't been modified then we will not be able to read from the right location in memory and might write to a wrong location. Hence, we require a lock on the symbol table.

References

[1] Mark-and-sweep collection (McCarthy, 1960)

[2] Jones and Lin, "Garbage Collection: Algorithms for Automatic Dynamic Memory", John Wiley Sons, 1996.

[3] LISP2 Algorithm, Wikipedia