OS Assignment 5 – Lab Report

Group 4

Esha Manideep Dinne – 19CS10030

ASRP Vyahruth – 19CS10002

Page Table Structure:

The page table is a simple array of integers which store the offset from the beginning for a given logical address, declared as ***int *page_table.*** This method of storing page_table gives us a way to use valid and invalid bits(first bit of every integer of the page_table).

Symbol Table Structure:

The symbol table has been implemented as a doubly linked list. The class structure is:

```
class s_entry {
    //0 - scope_elem, 1 - valid, 2 - markbit, 3 - isanarray
    bool specs[4];
    // bool scope_elem; bool valid;
    char type; char name[MAX_NAME_SIZE+1];
    int size; int logicaladdr;
    // bool markbit, isanarray;
    s_entry* prev; s_entry* next; //implemented as linked list
    int stack_entry;
public:
    s_entry();
    s_entry(char type_,char* name_,int addr,bool isarray_,int size_, bool scope_elem_ = false);

    //functions to get values

    void set_values(char type_,char* name_,int addr,bool isarray_,int size_, bool scope_elem_ = false);

    char get_type();
    char* get_name();
    int get_no_elements();
    int get_offset();
    int get_addr();
    int get_size();

    void add_offset(int x);

    void mark();
    void unmark();
    bool get_mark_bit();
    bool isarray();
    bool isvalid();
    bool doesexist();

    void set_stack_entry(int x);
    int get_stack_entry();

    void set_invalid();

    bool is_scope_elem();

    s_entry* get_prev_entry();
    s_entry* get_next_entry();

    void change_prev_entry(s_entry* tothis);
    void change_next_entry(s_entry* tothis);

    //debug function
    void print_entry();
};
```

bool specs [4] – a set of 4 specifications for each entry. First is if the entry is a scope element or not, third is markbit bit used while clearing the memory (to identify whether the entry is redundant or not), the second is valid to check if the entry is valid or not and the fourth is to identify if the entry is an array or not.

Char type – Specifies the type of each entry, whether it is an int, char, medium int (m_int) or bool

Char name [] – Stores the name of the entry (variable name / array name)

int Size – stores the size of the entry

int logical address – Stores the logical address of the entry

s_entry prev, next* – Pointer to next entry and previous entry in the symbol table (as we implemented the symbol table in the form a linked list)

int stack_entry – entry number of the stack. Needed to make sure that once an out of scope element is freed, it doesn't exist in the stack. We decided to add this instead of looping through the stack everytime we free an entry

change_prev_entry(s_entry tothis)* – Changes the previous entry. Used while changing the symbol table

change_next_entry(s_entry tothis)* – Changes the next entry. Used while changing the symbol table

All the other methods in the S_entry class are just get and set methods.

Memory Structure:

The memory has been implemented with a class myMem which is as follows:

```
class myMem {
    size_t size;  //size of the memory
    void *mem; //memory pointer

    int freed_mem;

    int next_mem;
    int* page_table; //stores the offset for a specific logical address

    pthread_mutex_t mutex;

    s_entry* first; s_entry* last; //pointers to first and last elements of the symbol table

    char* start_st_pointer;
    char* point_to_st;

    //private function to access the actual location in the allocated memory from logical address
    void* getphysicaladdress(int logical_addr);

    //get a free page for allocation or return the same page if a page is already allocated
    void* get_memory(int logical_addr);

    //check if the logical address is currently allocated or not
    bool isallocated(int logical_addr);

    void* get_location(char* name);
    s_entry* get_s_entry(char* name);

    void gc_initialize(); //add this in creatememm
    void gc_run();

    s_entry** var_stack;
    int var_stack_index;

    static void* wrapper(void* self);
    void pop_last_vars();

    void move_pages(char* from, char* to, int no_of_bytes = PAGE_SIZE);
    void updateLogicalAddresses(s_entry* first, int offset);

    void compact();

    int freeElem_gc(char* name);

public:
    myMem();
    myMem(size_t memsize);
```

```
public:
    myMem();
    myMem(size_t memsize);

    int createMem(size_t size);
    bool isempty();

    void* createVar(char type, char* name); //can be initialized with string as input
    void* createVar(int type, char* name); //can be initialized using the defined types as input

    template <typename T>
    int assignVar(char* name, T value); //to assign values to variables
    int m_int_assignVar(char* name, m_int value);

    void* createArr(char type, char* name, int size_);
    void* createArr(int type,char* name,int size_);

    template <typename T>
    int assignArr(char* name, T value, int index = -1); //index -1 means value is assigned to all elements of the array
    int m_int_assignArr(char* name, m_int value, int index = -1);

    //used for debug purposes
    // void* get_address(char* name);
    int freeElem(char* name);

    void scope_init();
    void scope_end();

    //debug functions
    void print_st();
    void display_empty_frames();
    void print_freed_entries();
    void call_compact();
    void print_final_freed_mem();

    void* gc_runner();
};
```

Size_t size – Size of our memory

void mem* – Starting Pointer to our 250MB memory

int next_mem – points to the next free location in memory

int freed_mem – amount of memory freed (used in hole percentage calculation which in turn is useful in deciding whether garbage collector should run or not)

int *page_table – our page table

*s_entry *first, last* – pointers to the first and last element of our symbol table (which has been implemented as a linked list)

char start_st_pointer* – a pointer to the next entry of the symbol table

Key Methods:

Void getphysicaladdress* (int logicaladdr) – Gets the physical address, given he logical address. The function shifts the logical addr by 2 bits to get the physical address.

Void get_memory(int logicaladdr)* – returns the page corresponding to the given logical addr if it exists or returns the new address to the logical address by allocating a page to it.

Void gc_initialize() – The function initializes the garbage collection thread.

Void* gc_runner() – This function runs an infinite loop. It calls gc_run() to run the garbage collection algorithm and sleeps for a given time and then loop again

Void gc_run() – The function that actually runs the garbage collection algorithm. Called both by the gc_runner() function and explicitly whenever we exit a function.

Static void wrapper(void* self)* – A wrapper function to call the garbage collection thread.

Void pop_last_vars() – pops the top element from the global variable stack

Void move_pages(…) – moves the pages during compaction

Void updateLogicalAddress(…) – updates the logical addresses during compaction

Void compact() – runs the compaction algorithm.

Int freeElem_gc(…) – Used by garbage collector to free pages from the memory

Void scope_init() – A special symbol will be added into the symbol table to mark the start of new scope.

Void scope_end() – Marks the end of the scope. All the elements till the next scope element are popped out and garbage runner is called.

Medium Int Structure:

The medium int has been stored as a char[3] array. Methods have been written to do mathematical operations on m_int, by converting this char[3] m_int into an integer and then return the value as another m_int.

Time of Garbage Collector to run:

We ran the demo1.cpp 10 times with garbage collector on and another 10 times with garbage collector off and noted down the average time the program took to run in both the cases.

Average time taken when garbage collector is ON – 5952.217 ms

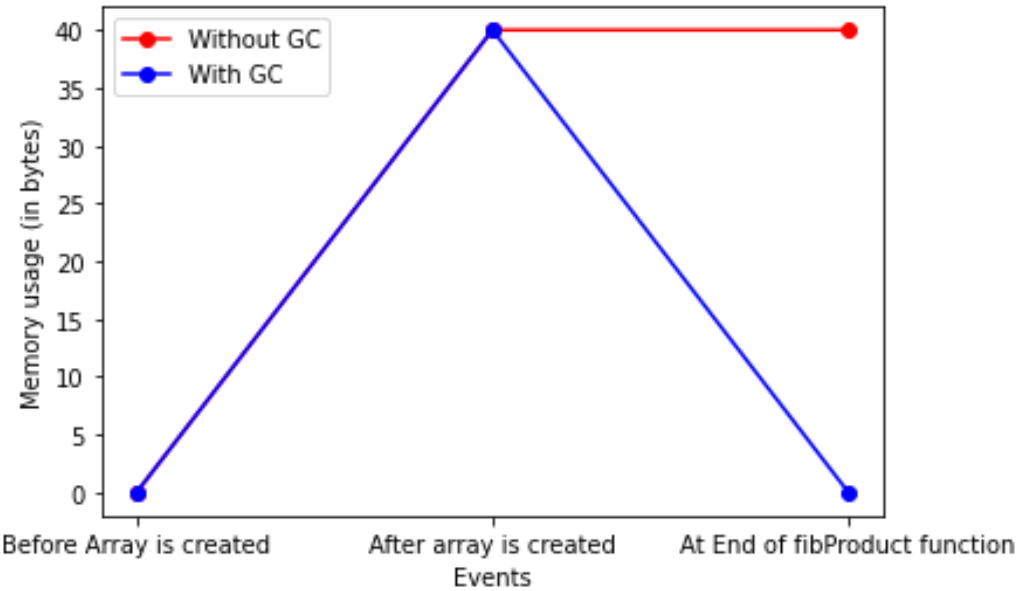
Average time taken when garbage collector is OFF – 5832.700 ms

Time of Garbage collector to run = 5952.217 ms – 5832.700 ms = 119.517 ms

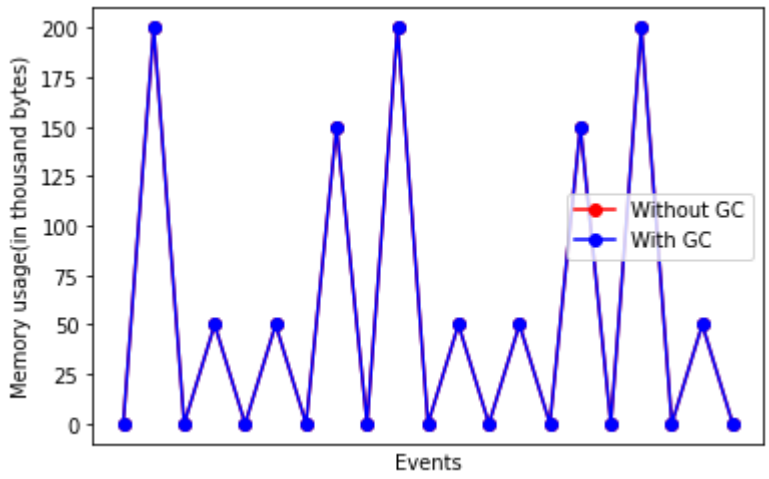
Memory Footprint of GC:

We ran the demo1.cpp with and without garbage collector. Garbage collector inherently doesn't take much memory(as it just uses the stack to manipulate the pointers). As we used first fit for allocating new memory, we noted down the hole space when GC is not run:

Total Hole Memory when Garbage Collector is OFF – 1150MB.



For demo1.cpp



There is no difference in memory footprint with or without GC because we are freeing the array before returning in the function. So here the main function of GC becomes management of holes as mentioned above.

LOGIC FOR COMPACT IN GC:

We first see total allocated memory. Then we loop through the entries of the page table. Whenever we encounter a hole, we find the next valid memory segment, move the whole remaining data to here. We continue doing this until we finished removing all the holes. This is the optimal algo, as we want contiguous memory allocation for arrays. If we don't want contiguous allocation for arrays, we could have done an O(n) algo but we decided to do this to keep arrays contiguous in memory. We invoke compact() from gc_run(). We wanted to invoke compact() when there are sufficient number of holes in the memory as it is expensive to run compact(). So, we defined a MAX_HOLE_PERC in the header file to denote the maximum hole percentage allowed before running compact. This is how we decide whether to run compact() or not.

LOCKS:

We used locks whenever we are creating a variable and whenever we are assigning a value, when we are freeing an element or when we are running Garbage Collection. Observe that as all of these need access to all Main Memory, Symbol Table, Stack, we need not separately lock these three data structures and instead we can lock all of them using the same lock. Lock is mainly used for gc_run() which is being run from another thread to avoid race conditions. All the createVar, assignVar, freeElem will be called sequentially by the user anyway. So the main race conditions the locks are preventing are the ones arising from

GC Thread. GC thread will cause race conditions when locks are not there. Suppose we are currently creating a Var, and it has been added to the symbol table, but before it gets added to the stack, context switch occurs and gc_run is invoked. Now, since the new entry is not in stack, gc removes it creating an issue. To prevent these types of scenarios, we decided to use locks.