OS LAB ASSIGNMENT-6

Manual Memory Management for Efficient Coding

Design Document

Group No: 17

Pranav Nyati (20CS30037)
Prerit Paliwal (20CS10046)
Vibhu (20CS10072)
Ashwani Kumar Kamal (20CS10011)

(A) Structure of Internal Page Table:

- **Basic Constructs, Definitions and Assumptions:**
    - Our implementation of the Paging consists of **fixed-size pages**: Each page is a multiple of size of 4 bytes (Size of int in 64-bit machine). Currently, each page has a size of **80 bytes** (can hold 20 int values or 20 4-byte values). The page size is a **macro** defined in **goodmalloc.h,** and can be varied to get different performance levels.
    - We have **NOT** used the **element** struct (like a node of a Linked list), and instead each of our list is a list of **INT** (for storing the values) , and no next and previous indexes. **Reason:** Our implementation has fixed size pages and each time a list is to be created, the required no of pages are allocated to it in the memory , and within each page, all elements are sequential, and for accessing values across pages, we have a vector to maintain sequence of pages allocated to each list.

- Our implementation of page table includes a struct **Sym_Table** and an ordered map **list_map**, which are explained in detail below:

● Sym_Table_Entry struct, Sym_Table struct and list_map:

- The Symbol Table has a fixed number of Symbol Table entries, defined by **NUM_SYM_TABLE_ENTRIES (which is 1,00,000)**, and can be adjusted.

- **List_Map:**
  - It is a **GLOBAL** ordered map with the key as **pair<scope(int), list_name(string)>**, and value as an index to one of the Sym_Table entries.
  - When a list with a particular name is created for a particular program scope, it first gets added to list_map, with the Key as (scope, list_name), and the value is an index to the newly created entry for that list (for that particular scope) in the Sym_Table.
  - When the scope ends, the entries for all Lists that were declared within that scope are deleted (in the **endScope()** function)

- **Sym_Table_Entry:**

```
typedef struct {

    unsigned long next_entry;
    unsigned long size;
    unsigned long capacity;
    vector< unsigned long> *list_pages;

} Sym_Table_Entry;
```

  - An instance of this struct is an Entry in the **Symbol table**. The **next_entry** is used to store the next free entry in the **Symbol Table** for a free entry and stores -1 for a filled entry.
  - The **capacity** is used to store the no of elements (integers in our case_ that the list can store.
  - **list_pages** is a pointer to a **vector <unsigned long>,** which stores sequentially all the pages in the memory allocated to the list corresponding to this entry. This vector gets populated when a list is created, and free pages available in the memory are assigned to it based on the requirements and cleared when the list goes out of scope (or if **freeList** is called for this list).

- **Sym_Table:**

```
typedef struct {

    size_t first_free_entry;
    size_t num_free_entries;
    Sym_Table_Entry *entries;

    void init();
    int addEntry(int scope, string list_name, vector<unsigned long> page_mapping, unsigned long capacity);
    void deleteEntry(int scope, string list_name);
    void printSymTable();
    void printEntry(int scope, string list_name);

}Sym_Table;
```

  - It is a **GLOBAL** table.
  - **Sym_Table_Entry *entries** is a pointer to an array of Sym_Table_entries and are dynamically allocated at run time.
  - **First_free_entry** stores the index of the first free entry in the **entries** array, and then the first entry stores the index of the next free entry, and so on.
  - **num_free_entries:** stores the no of free Sym_Table entries at any instant; if it is 0, no new list can be created, and the user gets an appropriate error
  - **init()** function is used to initialize the Sym_Table at the start of the program, and malloc the **entries** array and initialize it.
  - **addEntry()** enters the record for a new list created in a scope into **list_map** and also enters an entry for it in the **Sym_Table**
  - **deleteEntry()** is used to delete an Entry from the Sym_Table when a list's scope gets over (used in the **endScope()** function) or when freeList is explicitly called.
  - **printSymTable():** prints all Sym_Table entries , while **printEntry()** prints a specific one.

(B) Structure of Memory, Page Allocation Scheme, and other functions:

- **Structure of Mem_Block:**

```
typedef struct {

    int *mem;
    unsigned long capacity;

    //min heap of <int> to store the free pages
    priority_queue<int , vector<int>, greater<int>> free_list;
    unsigned long num_free_pages;

    void init(size_t size);
    void printMemPage(unsigned long page_num);
    void printFreeList();

} mem_Block;
```

- **mem** is a pointer (typecasted to int * for easy access to elements of a list) that stores the **base address to the initial large chunk of memory** requested by the user and is allocated using malloc
- **capacity** stores the size of mem in bytes
- **num_free_pages** stores the current no of free pages in the memory block, and the **free_list** stores the indexes of the actual free page nos in the memory in sorted (ascending order).
- **init()** is used to initialize the Memory block to the required size (to `size_t mem_size = (size_t)ceil((double)size/(double)PAGE_SIZE) * PAGE_SIZE;`) to get the memory in multiples of pages, and all pages to free_list initially
- **printMemPage()** is used to print the sequence of elements for that page and **printFreeList()** is used to print the index of all the free pages in memory currently

- ## Page Allocation Scheme:
  - The page allocation scheme is invoked when a call to **createList()** is made for a list of a particular name in scope. First, based on the requested size of the list, it is checked if the memory has enough free pages to allocate to the list; if not, an Insufficient Memory error is raised, and the program exits, else the page allocation scheme and starts allocating pages.
  - The list is allocated pages one at a time by popping a free page from the **free_list priority queue**, till the required no of pages are allocated. In this scheme, there is **NO EXTERNAL FRAGMENTATION**, as the allocation is done page by page, and even if there are many small small free segments

in the memory (consisting of a couple of pages), they will be allocated to a list if required.
    - Moreover, the **INTERNAL FRAGMENTATION** is also constrained by the **PAGE SIZE** (currently 80 bytes) and **can be a max of 76 bytes per list**.

- **Additional Functions:**
    - The basic functionalities asked in the problem statement: **createMem, createList, assignVal, freeList** have been written as it and are self-explanatory.
    - Apart from that, the **getVal(string list_name, unsigned long offset, int *value)** is similar to the assignVal function, just that it gets the value of a list at **position = offset** in the value (int pointer).
    - **compute_mem_address** is a helper function for getVal() and assignVal()
    - The **initScope() and endScope()** functions are used to maintain the scope information for a recursive function like **mergesort.** The initScope() is called at the start of a function definition, and endScope() and the end of the function just before the return statement. A **GLOBAL** counter **curr_scope** is maintained and is incremented each time initScope is called and decremented each time endScope() is called. Moreover, in endScope(), entries for all the lists declared in that scope are also removed from the Symbol Table.
    - **printList()** is a function to list sequentially all the pages corresponding to a list in the order in which they are allocated to it.

(C) Stats for MergeSort with and without freeList():

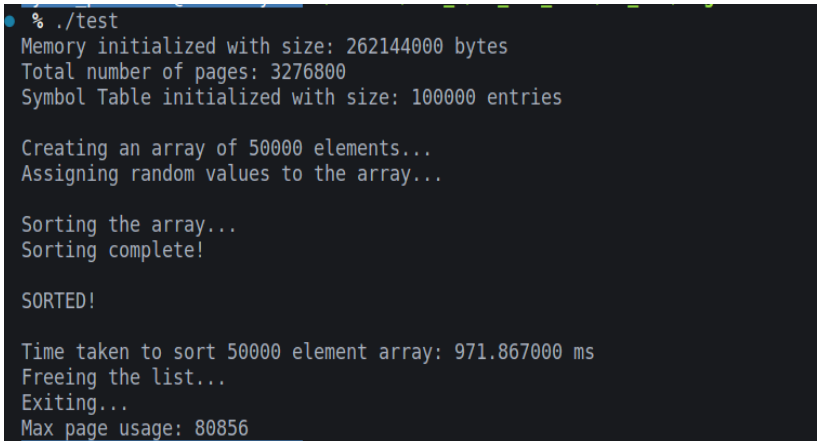
- **Impact of freeList() for MergeSort():**
    - The impact of calling freeList() in our implementation is **only with respect to the Memory Footprint of the process and not with respect to Running Time,** as we are allocating pages one at a time from a list of free pages, so searching for a free page is **O(1)** as it does not involve iterating through memory, and the initial memory allocation **(250 MB)** is very large, and even if one does not call **freeList()** on the lists, there are ample no of free pages in the memory to allocate to new lists.

- **Memory Footprint and Run-Time for the 2 cases:**
    - In the case when **freeList()** is **NOT CALLED**, the max no of pages occupied by all list during any instance of the program is **80856,**

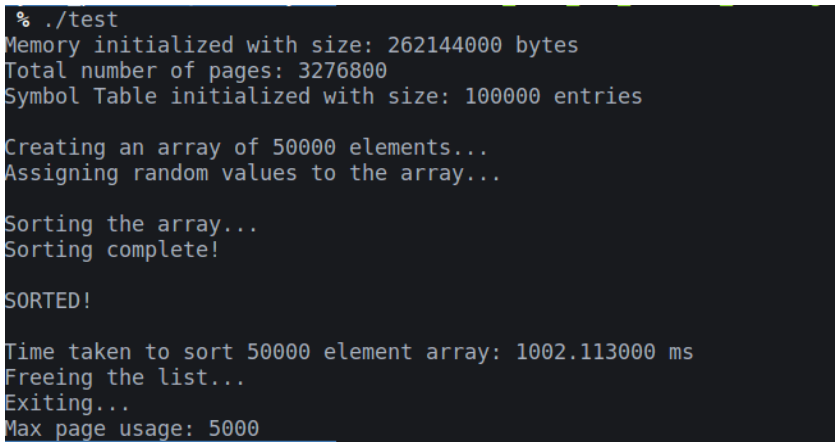
where the total no of pages in memory are **3276800 (250MB/SIZE_OF_ONE_PAGE = 80 bytes), which is approximately 2.47 % of the total memory.**

- In the case when **freeList()** is **CALLED**, the max no of pages occupied by all list during any instance of the program is **5000,** where the total no of pages in memory are **3276800 (250MB/SIZE_OF_ONE_PAGE = 80 bytes), which is approximately 0.15 % of the total memory.**
- Since both fractions are very small, the program's running time for the two cases will be very close **(around 1 second).**
  - **Average running time for 100 runs = 973.14 ms (without freeList())**
  - **Average running time for 100 runs = 967.55 ms (with freeList())**

- **Max_page_usage when freeList()** is **NOT CALLED: 80856 pages**



- **Max_page_usage when freeList()** is **CALLED: 5000 pages**



(D) In What Type of Code Structure is Performance Maximised, and where will it be Minimized?

- The performance will be MAXIMIZED for a code structure that DOES NOT HAVE a large number of recursive calls or multiple nested scopes of very large depths as if there are nested scopes (like in a function like Mergesort), a large part of the time is used in maintaining the scope and searching for the list (along with the scope) in the list_map as the list_map is implemented as a Tree Structure, and searching through it takes O(log(n)) time in the no of entries in the list_map, which will be more when there are nested scopes.
- With the above understanding, the Performance will be MINIMIZED for a function with a large no of nested scopes or a recursion tree of considerable depth involved in one or more of its functions. Similarly, performance will be MINIMIZED if the code involves creating too many small-sized lists rather than a few large-sized lists, as then bookkeeping time (maintaining the Sym_Table and scope handling) increases.

(E) Did You Use Locks in your Lib ? Why/Why not?

- No, we **DID NOT** use Locks in our Library as there are no multiple threads in our Library for Bookkeeping or Memory Management, as our design does not require parallel book-keeping or memory management, and any user program (like mergesort) that uses the Library uses static linking to that Library, and hence there are no multiple threads/processes sharing the memory space of the Library, hence no need for Locks.
