CS39002 - Operating Systems Laboratory

Assignment 6 DESIGN DOC

Group 10:
1) Anubhav Dhar - 20CS30004
2) Aritra Mitra - 20CS30006
3) Shiladitya De - 20CS30061
4) Subham Ghosh - 20CS10065

§1 Introduction

In this assignment we try to simulate the memory management via a C++ code supporting only one kind of data structure (a very specific linked list). We initially obtain a chunk of memory and manage, allocate & deallocate memory from this chunk. The details are as follows:

- `createMem()` takes in the number of pages that are to be used throughout. Can only be called once per process.
- `createList()` takes in the name and the total size of the list and returns the location of the head-pointer. We implemented our own `NULL` pointer named `NO_PTR` and it is returned in case of error.
- `assignVal()` takes in the name of the list, the offset and the value to be assigned. If no such list name exists, the list having the same name but declared in the nearest ancestor function would be taken instead.
- `freeElem()` deallocates the space of the input list name (or all list create in the caller function, if no arguments are provided). We used a trick to make sure that it is called when a function returns.

We discuss about other functions, the data structures used, the analysis behind the choice of design decisions taken and experimental results in the following sections.

§2 Page structure

Our pages in the page table are identical to elements of the linked list mentioned in the assignment. This design choice was made to remove external and internal fragmentation, and it performs optimally when there can be only a single type of element in the linked list.

```
struct element{
 long long val;           // eight bytes
 unsigned int prev_ptr;   // four  bytes
 unsigned int next_ptr;   // four  bytes
};
```

Implementation of the page

Note: In the above structure, we have used `long long val` instead of `int val` to store the data, it is because of the following two reasons:

- The total size of the struct with `int val` is 12 bytes (4 + 4 + 4) but gcc is going to pad extra 4 bytes to make it 16 bytes (so that it aligns with the 8 byte boundary in the memory reducing the no. of CPU cycles), so instead we used `long long int` which makes the total size 16 bytes.
- Secondly, having `long long int` also gives us a much greater range of numbers to work on.

We discuss the structure of the table (symbol tables) that maps list-name to pages later in the next section.

§3 Additional Data Structures and Functions

§3.1 Additional Data Structures

§3.1.1 list_meta_data

The `list_meta_data` structure stores the metadata about a list. Four things about the list are stored in this structure, as follows:

head_ptr: The head pointer *(offset in units of list element from the start of the memory)* of the list

curr_ptr: The pointer *(offset in units of list element from the start of the memory)* to the element in a list which was most recently accessed *initialized with head_ptr*.

idx: The index *(number of forward hops from the head_ptr)* of the list which was most recently accessed *initialized with 0*.

n: The number of elements in the list.

```
struct list_meta_data {
  unsigned int head_ptr;
  unsigned int curr_ptr;
  unsigned int idx;
  unsigned int n;
};
```

Implementation of list metadata

§3.1.2 symbol_tables

The symbol table is stored as `vector<map<string, list_meta_data>> symbol_tables`, where each element of this vector is a separate symbol table, and in each entry of a symbol table, a string *(name of the list)* is mapped to a `list_meta_data` structure *(which contains the metadata concerning that list)*. We are using this mapping/translation to access **pages** given the **list name**.

§3.1.3 caller_stack

The call stack is stored as `stack<pair<string, int>> caller_stack`, where each element of the stack is a pair, with the first element being a string *(name of the function)*, and the second element being an integer *(index in the `symbol_tables` vector)*.

§3.1.4 free_list

The list of free blocks are stored as `stack<unsigned int> free_list`, where each element is a pointer *(offset in units of list element from the start of the memory)*.

> ! It is stored as a stack as we would like to use, whenever possible, the memory location which was most recently used before it was set as free, as that has the highest probability of being cached, increasing the rate of cache hit, and making memory access faster.

§3.1.5 identifier_to_ST

The reverse mapping, from a defined identifier (list name) to the symbol tables where it is defined, is stored as `map<string, stack<int>> identifier_to_ST`, which each element maps a string *(name of the identifier)* to a **stack** of integers, where each integer is the index of a symbol table in `symbol_tables`.

> ! It is stored as a stack as we would like a function to use, whenever possible, the data that is created by its call chain, unless it has been rewritten *(which happens in case of recursive calls escpecially)*. As caller stack is a stack, the top of this stack denotes the closest ancestor function (and the corresponding symbol table) having the definition of this list name.

§3.2 Additional Functions

§3.2.1 readVal()

The `readVal()` function is used to access the values in a linked list. The function prototype is as follows:

```
int readVal(std::string list_name_string, unsigned int idx, long long * val);
```

This function takes the name of the list as a string, the index of the element in this list, and a pointer which whose pointed location will store the resultant value.
The return value is a return status, which has different values corresponding to different cases, as follows:

> **0 : SUCCESS**
>
> **1 : ERROR** The symbol table is empty.
>
> **2 : ERROR** The identifier is not found in the context of the caller function.
>
> **4 : ERROR** The specified index is out of bounds.

§3.2.2 freeElemReturn()

The `freeElemReturn()` function is the routine that happens when a function returns. The function prototype is as follows:

```
void freeElemReturn();
```

It calls the `freeElem()` function, removes the caller function from the call stack and removes the corresponding symbol table of that function.

§3.2.3 dbg()

The `dbg()` function is used for debugging purposes. The prototype is as follows:

```
void dbg(int line);
```

When called, this prints all the symbol tables at that point of execution, and also the mapping of identifier to symbol tables.

§4 Impact of `freeElem()` on Mergesort

§4.1 Running Times:

In this section we are comparing the running times of mergesort when `freeElem()` is called explicitly and when it is not called from the mergesort function.

§4.1.1 If `freeElem()` is called explicitly from the mergesort function

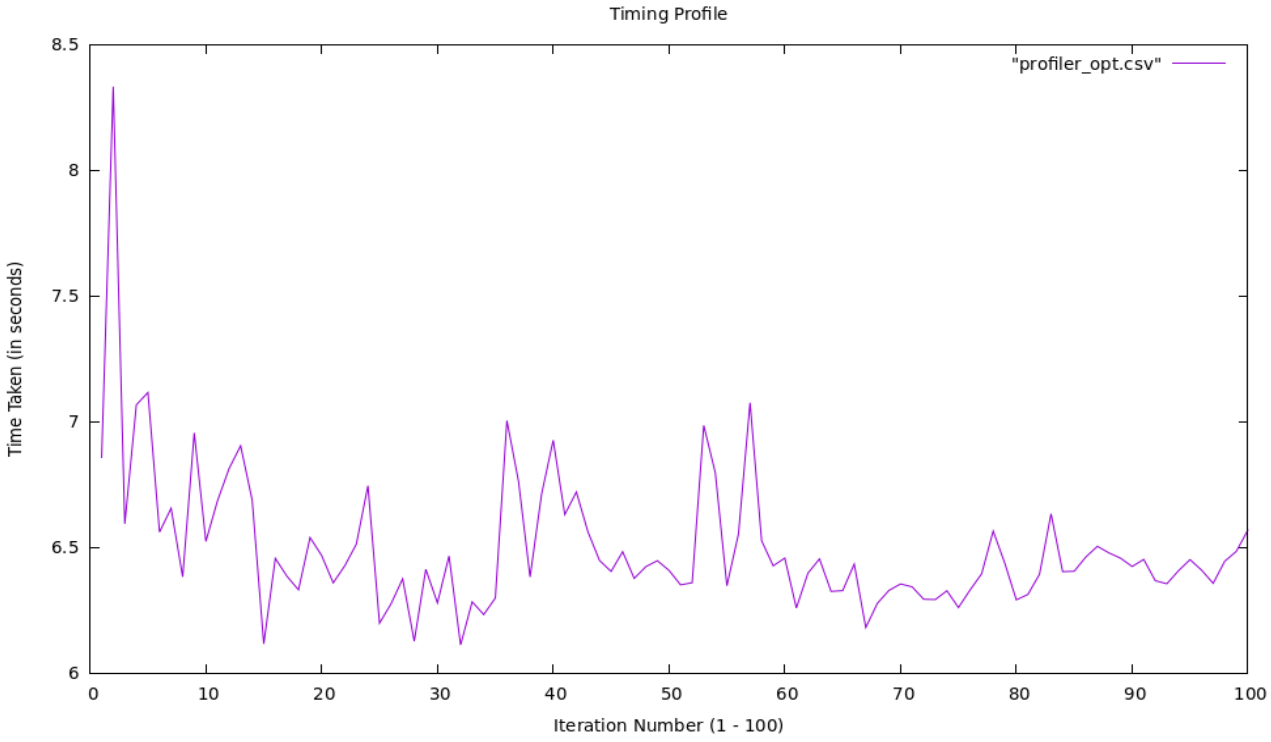
In this case the `freeElem()` function is called explicitly even though the return statement has been tweaked through the usage of a `#define` macro as follows:

```
#define return caller_func=strdup(__func__);freeElemReturn();return
```

The metrics generated are as follows:

```
Mean: 6.49811 seconds
Standard Deviation: 0.28165
Coefficient Of variation: 4.33446 %
Median: 6.43073 seconds
```

The plot of the timings of 100 runs of mergesort:

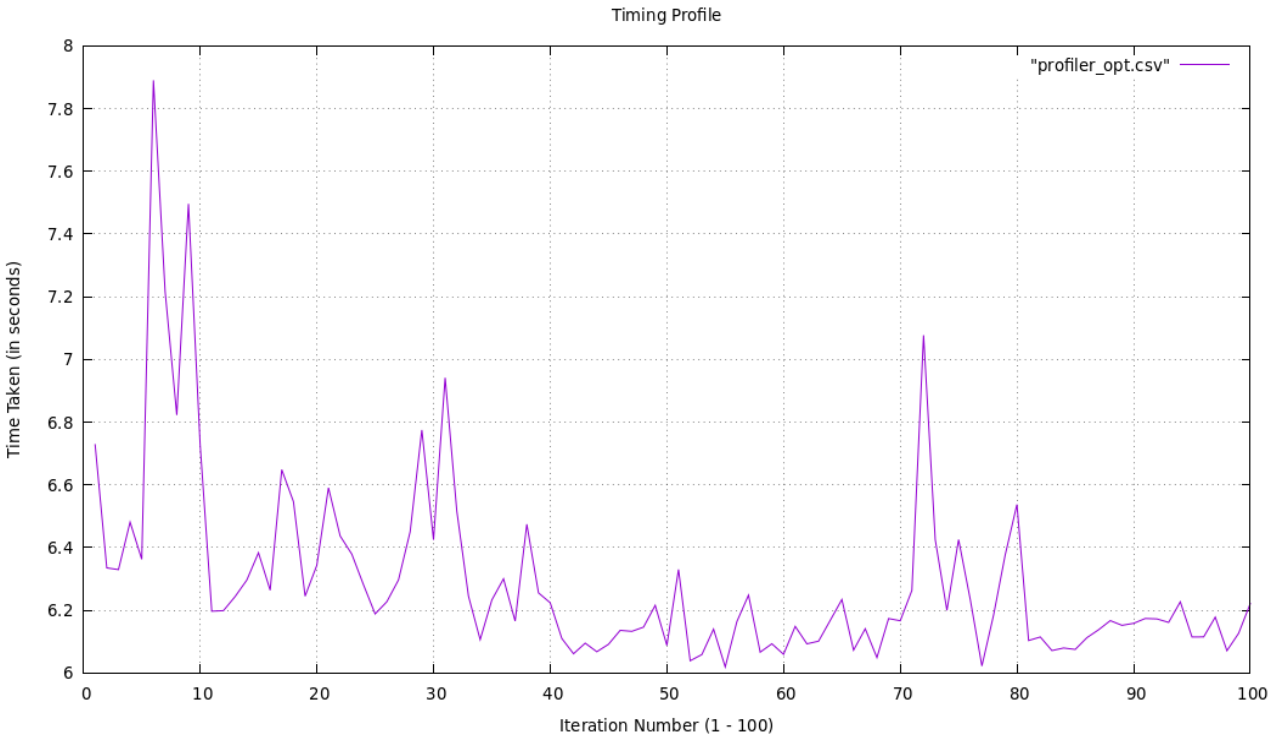


§4.1.2 If `freeElem()` is not called explicitly from the mergesort function

In this case, `freeElem()` is not called explicitly. The metrics generated are as follows:

```
Mean: 6.29469 seconds
Standard Deviation: 0.30164
Coefficient Of variation: 4.79203 %
Median: 6.19945 seconds
```

The plot of the timings of 100 runs of mergesort:



In both the cases the initial spikes are mainly due to cache misses and random values of the input array. The numbers do not vary much as we had triggered `freeElem()` from return itself, hence it is called anyway in both cases.

§4.2 Memory Footprint:

§4.2.1 With `freeElem()`:

When `freeElem()` is called then effectively after each recursive function call the memory is freed. Hence at the end, we are going to get one array of length 50000 and two arrays of length 25000 each. Thus the memory taken is $2 \cdot 50000 \cdot 16$ bytes $= 1.525$ MB.

§4.2.2 Without `freeElem()`:

When `freeElem()` is not called then the memory is not freed. Since we have $\lceil log_2(50000) \rceil$ levels in the recursion tree which gives us a total memory requirement of $50000 \cdot \lceil log_2(50000) \rceil \cdot 16$ bytes $= 12.20$ MB.
However, since our code always calls `freeElem()` irrespective of the user calling it explicitly or not, the memory footprint will be the same as with `freeElem()` which is 1.525 MB.

§5 Conditions on Performance

§5.1 Optimal Performance Criteria

Our implementation works best if the following conditions hold, and each of them individually help the performance:

> **Less Functions** If the number of functions called during the execution of the code is lower, then our code will function better as the `freeElemReturn()` function will be called less times, as well as we would have less number of symbol tables, smaller caller stack and less bookkeeping in general, resulting in less overhead.

Sequential Access If the elements of the lists are accessed sequentially, then this implementation will work best, as the seek inside the list starts from the previously fetched index, making sequential access fast.

Less Functions without list If the number of functions without list usage called during the execution of the code is lower, then our code will function better as there will be less number of calls of `freeElemReturn()` which removes no list from symbol table.

§5.2 Suboptimal Performance Criteria

Our implementation works suboptimally if the following conditions hold, and each of them individually hurt the performance:

More Functions If the number of functions called during the execution of the code is higher, then our code will function suboptimally as the `freeElemReturn()` function will be called more times, as well as we would have increased number of symbol tables, larger caller stack and more bookkeeping in general, resulting in more overhead.

Sequential Access If the elements of the lists are accessed randomly, then this implementation will work poorly, as the list is essentially sought sequentially, making random access slow.

More Functions without list If the number of functions without list usage called during the execution of the code is higher, then our code will function suboptimally as there will be more number of calls of `freeElemReturn()` which removes no list from symbol table.

§6 Usage of Locks

No locks has been used in this assignment because there are no multiple threads or processes involved which can create a synchronisation problem.

§7 Triggering Returns & Corresponding User Code Requirements

As we have already discussed, if we need to trigeer something when some functions return, (call `freeElem()` in this case) we need to change the compiler entirely. As this is not possible, we use the trick of `#define` to implement the triggering. However, this adds on minor restrictions on the code. These are:

- The user must write `return` even for functions that return `void`.
- Any `return`, `createList`, `freeElem` calls must be inside a function or a scope.
- Similar to the old versions of gcc, the `createList`'s must be done in the beginning of the function.