Operating Systems Laboratory (CS39002)

Assignment - 6

Spring Semester 2022-23

DESIGN DOCUMENT:

a) Data Structures for memory storage and creation:

```
extern struct Element *my_memory;
struct Element
{
   int value;
   Element *prev = NULL;
   Element *next = NULL;
};

// Managing Paging
set<Element *> free_pages; → used in createList() to find free pages in
O(log(PAGE_COUNT)) time
stack<set<Element *>> occupancy → stack of sets where the topmost set maintains the
variables (lists) being created in the current function call
```

Memory Creation using createMem()

- The createMem(int size) function creates a contiguous memory of *size* rounded up to the nearest multiple of PAGE_SIZE.
- We implement Paging as an abstraction provided by our library over the above created memory.

Why use paging and choice of PAGE_SIZE?

- A naive implementation without use of paging would result in 2 scenarios.
    a. If linked lists are created according to a policy that allocates contiguous memory blocks (so as to enable random access in O(1) time), this would result in **external fragmentation** increasing the memory footprint.
    b. If contiguity was not preserved, this would lead to random access of elements in O(n) time as in worst case we would have to iterate over the next pointers of all elements.

- We achieve an efficient solution using paging that prevents external fragmentation. Suppose a linked list occupies k pages, these pages may be non-contiguous but may be linked by the next pointer of their last elements. Within a page the elements of a list may

be accessed in constant time. Thus random access as a whole is achieved in O(n/PAGE_SIZE)

- We appropriately choose **PAGE_SIZE=256** to achieve a optimum trade-off between running time and reduction in internal fragmentation

- PAGE SIZE and time comparison, footprint

b) We provide the following function calls as a part of our library
   a. void createMem(int size);
   b. Element *createList(int length, Element *&listName);
   c. int assignVal(Element *listName, int offset, int value);
   d. Element *accessElem(Element *listName, int offset);
   e. void freeElem(Element *listName);

   We provide the additional function ***accessElem()*** to provide efficient random access to the elements in the list returned by createList()

c) **freeElem()** ensures space (pages) occupied by the unused lists are freed before returning the function. This ensures that the active number of pages at any time does not explode, which is crucial especially when multiple recursive calls are made in a function like mergesort().

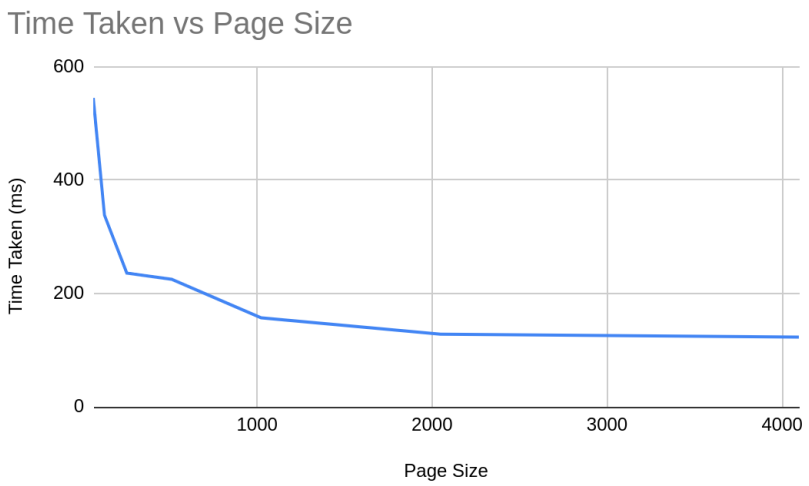
Following Table compares merge sort code with and without freeElem():

Mode	Time Taken (ms)	Max Number of Pages
without freeElem()	572	51672
with freeElem()	**236**	**392**

We see a staggering reduction of memory footprint of over 10x using freeElem() and 2x improvement in total run time.

Comparison of Page Size vs Time Taken

Page Size	Time Taken (ms)
64	545
128	338
256	**236**
512	225
1024	157
2048	128
4096	123



d) **Performance**: Memory wasted due to internal fragmentation is minimized when the size of list requested by the user programme is just less than or equal to the PAGE_SIZE, and it is maximized when the size is just greater than the PAGE_SIZE. This is because the remainder of the page is left unused and cannot be allocated to any other list.

e) We do not use any locks in our library as implementing locks at the level of the memory management software prevents parallelization at the level of the user. It is thus left to the user to implement locks to manage-shared access to the memory.