Operating Systems Lab
Assignment 6 - Memory Management
Spring 2023

Group Members:
Anand Manojkumar Parikh (20CS10007)
Soni Aditya Bharatbhai (20CS10060)
Pranav Mehrotra (20CS10085)
Saransh Sharma (20CS30065)

Manual:

- int createMem(int size) –
  Dynamic memory is allocated of size equal to the next largest (greater than or equal to) multiple of PAGE_SIZE (256) bytes.
- int destroyMem() –
  Deallocates memory allocated by previous createMem call.
- int createList(string name , int size) –
  Creates a list with the name "name" of "size" bytes.
- int assignVal(string name , int offset , int val) –
  Assigns value "val" to 4 bytes of memory starting at "offset" byte in the list named "name".
- int readVal(string name , int offset , int * val) –
  Retrieves a value of 4 bytes starting at "offset" in the list named "name" into the integer pointed to by val.
- int freeList(string name) –
  Deallocates the list named "name". If no argument is provided (overloaded function is called), then name is defaulted to empty string and all lists (if any) in the current scope are deallocated.
- int scope_start() –
  User is responsible for appropriately calling start_scope() to notify the library of a new scope or recursive call (not mandatory in all cases, depending on usage).
- int scope_end() –
  User is responsible for appropriately calling end_scope() to notify the library of the end of the current scope (not mandatory in all cases, depending on usage).

The above function calls return 0 on success. On failure, -1 is returned and ERRNO is set indicating the error type.

ERRORS:

- SCOPE_ERR –
  A call to scope_end() makes the scope value negative.
- MEM_ERR –
  destroy_mem() was called when no memory was allocated by createMem().
  create_mem() was called without destroying previous createMem() memory.
- SIZE_ERR –
  create_mem() or create_list() are called with either a non-positive or an excessively large "size" argument and allocation failed.
  assignVal() or readVal() are called with either a negative "size" argument or a "size" argument such that placing / fetching the value would access memory not within the bounds of the list specified by "name".
- DUPLICATE_ERR –
  createList() was called with a "name" previously used within the same scope without freeing it.
- LIST_NOT_FOUND_ERR –
  Any operation was called on an unknown list name.
- NAME_ERR –
  createList() was called with an empty string in the "size" argument. This is not allowed, since "" is a system-specific default list name used to detect overloaded usage of freeList().

Internal data-structures:

1. class List{
   ```
   string name;    // name of the List in string format
   int scope;      // scope value when it was created
   int size;       // size (in bytes)
   vector<char *> PageTable;   // explained below
   ```
   }

   A List may span multiple pages in the memory, so each List has its own page table, which is simply a list of the starting addresses of its pages in the buffer allocated using createMem(). An address is in the format of <char *> to make the memory byte-addressable.

2. map<pair<string,int> , List> Lists;
   The unique identifier of a list is the <name , scope>, since the same name may be reused in different scope. So a std::map of std::pair<string , int> to a List

object is used. [unordered_map is avoided since it may lead to collisions and hence bad worst-case time complexities]

3. set<char *> freePages;
   The only information needed about a page is its starting address, since the page size is already fixed. So, all pages are indexed by their starting address <char *> and pages which are free to be assigned to a list are stored in a std::set, since each page has a unique starting address in the buffer. [unordered_map avoided for the same reason]

Page allocation policy:

A mixture of best-fit and first-fit is used to allocate pages to a List by the following steps-

1. First, check if total available memory (number of free pages * page size) is greater than or equal to the requested bytes. If not, raise error. Else, go to step 2.
2. Find the largest contiguous block of available pages. If that satisfies the request fully, finish with success. Else, allocate the block of contiguous pages, remove those pages from the list of available pages and go to step 2.

Impact of FreeElem() for MergeSort():

In the merge sort code the FreeElem plays a crucial role, as if not used the memory footprint exponentially increases because of the recursive nature of mergesort.
In our case, we called CreateList and FreeElem in all the merge steps and in all merge steps, 2 lists are created and then freed. So, at any point during the execution of our mergesort, there are at max only 3 lists, 1 the global list from Main and 2 local lists which are created and deleted on the same level of recursion. So, the maximum amount of memory being used at any point of execution, would be at the topmost level when merging the 2 sorted lists of 25000 integers each. Hence at the topmost level, the maximum number of pages allocated at the same time is **1564 pages** of 256 bytes each, meaning a total of **400,384 bytes.**

However, if we don't call FreeElem(), then the allocated number of pages keeps increasing through the recursion and leads to allot of memory usage. In this case the total number of pages allocated is **107,468 pages** of 256 bytes each, meaning a total of **27,511,808 bytes**. Which is **68.714 times** the memory used than the case when FreeElem was called after each merge step.

Optimum Code Structure

So, as clearly evident from previous arguments, if the memory is not cleared between recursion steps, it can lead to poor performance and large memory footprint. Therefore,

a code structure in which the Lists are created and freed before the invocation of recursive call or after the return of recursive call, then the memory footprint would be limited and acceptable.
However, if the lists are created and then the recursive steps are invoked and the lists are freed after the recursion returns. Then the memory footprint can exponentially increase and lead to poor performance.
Therefore, if lists are nor freed before the recursive calls, the performance would be **minimised**, where as if lists are only created befor or after recursive calls, the performance would be **maximised**.

Justification of design choices:

1. Splitting memory in pages: This is done to allow direct access within a page. The page number can be found using the quotient of dividing offset by page size. The remainder gives the offset within the page. This allows very fast direct access with only 1 extra stage of indirection.
2. Page size = 256 bytes: Having a very small page size will incur a very heavy memory overhead on the memory manager, since now more addresses need to be recorded. On the other hand, using a very large page size leads to internal fragmentation (especially in the mergesort application, where the final recursion step has only at most 2 integers).
3. Not using pointers / chaining anywhere: Using pointers is very wasteful for 2 reasons-
    a. Memory overhead of storing these pointers - The idea of storing 2 pointers (2 bytes each), say for each 4 byte integer incurs overhead = allocated space, reducing memory efficiency to 50%
    b. Slow sequential access - Access to any memory location now becomes sequential. This leads to a drastic speed degradation, both if linked / chained lists are used to chain pages or to chain individual elements (it becomes totally unacceptable at the element level).

Additional questions:

1. Were locks used in this library?
   No locks have been used to implement this library, simply because they are not needed. It is a single-threaded library. It can, however, be used in a multi-threaded environment, but it is the user's responsibility to enforce mutual exclusion (possibly using user-space locks, totally up to the user) and mark scope start/end appropriately. The library inherently does not recognize the existence of threads.