Operating Systems Laboratory (CS39002)

Spring Semester 2021-2022

Assignment 6: Implement manual memory management for efficient coding

Assignment given on:	29th March, 2022
Assignment deadline:	12th April, 2022, 11:55 pm
Total marks:	100

This assignment is related to virtual memory management inside the operating system. In general, in many large-scale real-world systems (e.g., database servers) the software does not depend on the OS to do memory management. In general these softwares requests (right at the beginning) a chunk of memory using malloc (e.g.,500 MB, 1 GB). Then the customized memory management module for these softwares manages the memory (e.g., assigning memory, freeing it up etc.).

To that end, you will implement and learn a basic memory management module. Here are your step-by-step tasks for a C/C++ code you need to write.

1. Create your memory management library — a header file "goodmalloc.h" and an implementation file "goodmalloc.c"

2. **Functions of your library:** The library provide you (at the minimum) the following functionalities (more details in the next points):

   a. createMem – a function to create a memory segment; your code written using this library will use this function at the beginning to allocate the space. You can think of the returned memory as an initial big array of memory blocks.

   b. createList – using this function, you can create a linked list of *element* . You can think of each *element* as a structure with one integer and two indexes holding addresses of previous and next elements in the list, i.e, pointers (similar to head and tail pointers of a doubly linked list). These variables will reside in the memory created by createMem. Note that you need to *create* and maintain this special data type and set the pointers correctly.

      createList takes in two arguments—size of the list and name of the list. So your code written using your memory manager function can only use these linked lists of elements.

      Every time a createList function is encountered, a new local address for the list is generated which maps the list name to the start address of the linked list. During updation etc. this local address is then converted to a suitable address within the assigned memory to access the data sequentially.

Use the concepts learned in paging and page tables. Implement First Fit or Best Fit to find a valid free segment of the local address space to store new lists. If there is no free segment which is big enough then print an error and exit.

Additionally, a function might contain local lists which are not needed when a function returns, just before returning you need to do some bookkeeping to free the memory, namely popping the local variables from a global stack (to keep track of which memory is not needed any more) and freeing up memory..

c. assignVal – assign values to the *element* lists in your locally managed memory. As input, take in the name of the list, offset of the starting element to be updated, number of the elements to be updated and an array of values. Returns error if the size of array does not match the number of elements to be updated.

d. freeElem – using this function you can free the memory occupied by lists when they are no longer used (e.g., freeing local lists when a function returns).
   - freeElem should be called to free up space for local variables of a function right before return and should be run whenever explicitly called.
   - Naturally freeElem will update your local page table.
   - Calling freeElem on an invalid local address should raise an error and exit.

Print out messages in the terminal when you are calling library functions and/or creating/updating local page table entries.

3. **Now write code with your library:** To demonstrate your library's functionality write a code mergesort.c using calls from the library where applicable.
   a. mergesort.c : Your main function should take 250 MB memory. Then it should create a linked list of 50,000 elements and fill the values with random integers between 1 and 100,000. Next, you will sort this list using a recursive merge sort code. In the recursive call/ helper functions you can use additional lists created by the createList call. Note that, to support recursion on heap, you need to scope the variable names, as different calls to the same

function would have the same variable name, that needs to be differentiated (you might need a stack of your own to keep track).

4. **Write a report:** Note that by now you have made a lot of design decisions in your code (e.g., what data structures to use). You now need to describe and justify them:

    a. What is the structure of your internal page table? Why?

    b. What are additional data structures/functions used in your library? Describe all with justifications.

    c. What is the impact of freeElem() for merge sort. Report the memory footprint and running time with and without freeElem (average over 100 runs).

    d. In what type of code structure will this performance be maximized, where will it be minimized? Why?

    e. Did you use locks in your library? Why or why not?

5. **Submission:** Submit a “assignment6_GroupNo.zip” with the following files:
    a. goodmalloc.h, goodmalloc.c, mergesort.c, Makefile (to compile and run mergesort), report.pdf

6. **Marks distribution**

	marks
createMem	5
createList	15
assignVal	15
freeElem	25
Any other functions	15
mergesort.c	15
Report	10