Operating Systems Laboratory (CS39002)

Spring Semester 2021-2022

Assignment 5: Hands-on experience for creating better memory management systems

Assignment given on: 15th March, 2022

Assignment deadline: 29th March, 2022, 1:00 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 for managing more context-specific memory (e.g., storing special bits in the page table). In general these softwares request (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 manage 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 "memlab.h" and an implementation file "memlab.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 using malloc, your code written using this library will use this function at the beginning to allocate the space.

    b. createVar – using this function you can create a variable. Consider the following variable types – int (4 byte), char (1 byte), medium int (3 byte), boolean (1 bit). These variables will reside in the memory created by createMem.

    Every time a create variable/array function is encountered, a new local address for the variable is generated which is basically an internal counter. During updation etc. this local address is then converted to a suitable address within the assigned memory to access the data. Use the concepts learned in paging and page table. Implement First Fit or Best Fit to find a valid free segment of the local address space to store new variables. If there is no free segment which is big enough then print an error and exit.

Additionally, a function contain local variables which are not needed when a function return, just before returning you need to do some book keeping, namely popping the local variables from a global stack using garbage collector and free up memory, more details are in garbage collection section.

c. assignVar – assign values to variables. Have a light type-checking, e.g., your boolean variable cannot hold an int.

   For variable access and modifications, find the local address, then the logical and then access the page table, locate the corresponding frame, access or modify the content. Keep track of the size of the variable for proper access.

d. createArr – using this function you can create an array of the above types. These variables reside in the memory created by createMem.

e. assignArr – assign values to array or array elements. Have a light type-checking, e.g., your boolean variable cannot hold an int.

f. freeElem – using this function you can free the memory used by the variables and arrays when they are no longer used (e.g., local variables when a function returns). You also need to implement garbage collection (see below).

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

3. **Word alignment:** This is an extremely useful concept. Often the memory is read in chunks of words (not bit by bit or byte by byte). So, facilitate speed-of-access in your library, all memory accesses will happen in a word aligned manner. One word = 4 bytes.

   Thus, your local address (the counter value as used internally by your library) will always be a multiple of 4. If a previous variable is not of size which is a multiple of 4, just allocate redundant space.

   Also, you need to track the actual size of the variable is necessary for proper access and modification of variables.

   There can be a helper function for accessing variable bytes of data, but the final interaction with the assigned memory (created by createMem function) must access data chunks of byte 4.

   Print out messages in the terminal when you are doing word alignment.

4. **Garbage collection:** Garbage collection is just a fancy name for freeing up memory which is not used opportunistically. Your createMem gives you a limited amount of memory, so you need to efficiently use this memory (and free it up whenever not needed)

    - Since this a basic assignment, we will only consider a serial garbage collection (runs in one thread, good for programs with small memory footprint)

    - Run a *garbage collection* thread (by calling gc_initialize first and then gc_run in subsequent calls) periodically to free up memory allocated to variables or arrays which are not used any more.

    - Use a *mark and sweep* algorithm. You will keep track of the variable references in your user space code using a stack (part of your library). When a function returns you will pop the variables used from the stack by calling a function in your library.

    - Your garbage collector will run periodically (as well as just before returning from a function) to check the global stack (*mark phase*) and then go to your memory space to free up the unused memory (*sweep phase*) after consulting the page table.

    - A final modification is a slight variation of the mark and sweep algorithm to *compact* during the sweep phase. Keep pointers to the freed up and used space in your garbage collector. If your memory address space (set of counters) contains *large holes* after freeing up memory, then copy those memory locations to make the used memory as contiguous as possible.

    Print out messages in the terminal when you are doing garbage collection (showing different steps of gargabe collection).

5. **Using locks:** Because garbage collector runs in a thread and effectively update shared data structures you might want to use locks to stop data corruption.

6. **Now write code with your library:** To demonstrate your library's functionality write code files demo1.c, demo2.c using calls from library where applicable.

a. demo1.c : Your main function should take 250 MB memory. Then it should call 10 functions with two parameters x and y (x and y same data type). Each function will create and populate an array of 50000 elements of the same data type with random data, destroy the array and return.

b. demo2.c : Implement a fibonacci function in your code using memory management. Then your main will take a parameter k, and pass it to a function fibonacciProduct which will call the fibonacci to populate an array of first-k fibonacci numbers, compute its product and return the product.

7. **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 mark and sweep garbage collection for demo1 and demo2. Report the memory footprint with and without Garbage collection. Report the time of Garbage collection to run.

   d. What is your logic for running compact in Garbage collection, why?

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

8. **Submission:** Submit a "assignment5_GroupNo.zip" with the following files:
   a. memlab.h, memlab.c, demo1.c, demo2.c, Makefile (to compile and run demo1 and demo2), report.pdf