CS39002: Operating Systems Lab
Spring 2012

Assignment 7
Due: March 22, 2012, 1 pm

Part 1: Designing a personal memory-management library

In this part, you design a user-level memory-manager library for C as follows. When you write a C program, you use *malloc()*, *free()* etc. to allocate/free memory. With the new library that you design here, your C programs will not use *malloc / free* directly, but will instead call the functions provided by your library to allocate/free memory. So the output of this part is a static library *mem_mgmt* that contains the following list of functions for memory management. The functions work as described below.

- *int init_mem()*: This should be the first function to be called by any C program that uses your memory manager. This initializes the data structures necessary for the memory manager. It also creates a chunk of memory to manage of size 16 KB (this is to be created by a standard *malloc()* call). All further allocations will be done from this chunk of memory, which we henceforth refer to as *the personal memory pool*. The function should return 1 in case of error, 0 otherwise.
- *void * mem_malloc(int size)*: This function allocates memory of size *size* bytes from the personal memory pool created and managed by the memory manager. If allocation is successful, address of the first location is returned; if not, NULL is returned.
- *void mem_free(void *p)*: This function frees a memory pointed to by *p* and allocated earlier by a *mem_malloc()* call. The freed memory is returned to the personal memory pool.
- *void mem_stat()*: This function displays the total allocation from the personal memory pool and the amount of free space left in the pool, the blocks allocated (starting address, size), and the free blocks.

You should use the *buddy system* to manage the memory. Assume that the minimum request size will be for 16 bytes (if a request is for less than this, allocate it 16 bytes at least).

The memory manager should flag an error if allocation/freeing is attempted before initializing the memory manager (that is, before calling *init_mem()*). You may write any other internal function that you may need.

Create an appropriate header file *mem_mgmt.h* to store all type definitions and function prototypes.

Part 2: A test program

First, test your library by writing a C program *mem_test.c* that first initializes the memory manager, and then randomly allocates/frees memory. Use *mem_stat()* to verify that the memory looks fine. Check for boundary cases, like trying to allocate when memory is full, trying to free non-existent memory, etc.

Remember that the functions in Part 1 should be made into a library that will be linked to your C program *mem_test.c*; they should NOT be additional functions defined in *mem_test.c* or, for that matter, in any application program that uses the library. Moreover, the declaration, initial allocation, and management of the personal memory pool should remain transparent to any application program. It is as if the standard *malloc* and *free* calls are replaced by *mem_alloc* and *mem_free*.

Part 3: A sample application program

Write a program *sort_merge.c* that does the following. The program first initializes the personal memory manager. It then creates two worker threads which read two files *input1.txt* and *input2.txt* storing two unsorted lists of integers (with a few hundred entries each). Each thread creates a binary search tree in the personal memory pool for storing the integers listed in the respective input file. Each node in the BSTs should consist of a key field and two pointers (left and right). These pointers are allocated space in the personal memory pool by invoking the *mem_alloc()* function.

When each thread has completely read its input file and stored all the keys read in its BST, it creates an array, again in the personal memory pool. The allocation size of the array must be exactly equal to that needed for storing all the keys in its BST. The thread then makes an in-order traversal of its BST, and stores the sorted sequence of keys in its array. The BST is then freed by invoking the *mem_free()* call node by node.

The threads terminate after creating two sorted arrays in the personal memory pool. After this, the master thread merges the two sorted arrays (avoiding duplicates), and stores the merged list to a file *output.txt*. The master thread finally frees the space allocated to the two sorted arrays, and exits.

Submit the following files in a tar-ball.

1. *mem_mgmt.c* (containing the source code for your library)
2. *mem_mgmt.h* (the header file to be included by application programs)
3. *mem_test.c* (application program for testing the functionality of your library)
4. *sort_merge.c* (application program for sorting and merging two files)
5. A *makefile* for compiling your library and the two application programs

---