TCSS333 - C for System Programming
Programming Assignment 7
Simulated Heap Using Linked Lists

DUE: See Canvas Programming Assignment 7 link.

Create a file called mallok.h The contents of that file should be:

Conditional preprocessor command of #ifndef MALLOK_H and #define MALLOK_H, all necessary include directives, data structure of your list nodes and list pointer, followed by the following function prototypes:

```
void create_pool(int size);
void *my_malloc(int size);
void my_free(void *block);
void free_pool();
```

ending with an #endif pre-processor directive.

You may not modify these prototypes. The work you submit will be tested using a secret grader program written to compile with this header. If you modify these prototypes at all (including the parameter descriptions or the capitalization or spelling of the function names) the grader's program may not compile and your work will be penalized.

Your implementation will be an "abstract object", not an ADT as discussed in Chapter 19. In other words, you will be programming a single global heap, not a heap class. You may use a small number of global variables to implement your heap, but these should be marked "static" to limit access to them. The argument for create_pool indicates how many bytes should be included in the heap. Your implementation of create_pool must use malloc to acquire a single block of exactly this size. This single block is your "private heap". All calls to my_malloc must be satisfied by returning a piece of this private heap if possible. my_malloc should not simply use malloc to allocate the block the caller has asked for. Likewise, my _free should return a block to the private heap, not to the built-in heap by calling free. my_malloc and my_free have the same parameters as malloc and free (although the types have been simplified just a little).

Initially the private heap contains a single free block that has the same size as the call to create_pool. In responding to my_malloc and my_free calls, this initial block will sometimes be split into smaller pieces and these pieces must later be recombined (coalesce) whenever possible. At any given time, some of the blocks will be in use because my_malloc has given them to a caller, or they will be free, meaning they are (back) in the private heap. You must build a single linked list to keep track of all the blocks, whether they are free or not. **The linked list will be similar to the list you built in HW6, but each node will keep track of a storage block taken from the private heap. For each block, be sure to remember - where it starts (a memory address) - how big it is (a number of bytes) - whether it is free or not. Your linked list of blocks must be kept in order by the starting address of the block.** This means that blocks that are next to each other in memory will also be next to each other in your linked list of blocks, i.e. you need the capability of inserting nodes "anywhere" within your list. To build your linked list of blocks, you will need to create list nodes. You ARE allowed to use malloc in the usual way to create a linked list node. (But as stated above, you are NOT allowed to use malloc to create the blocks themselves.) What is described above implies that this list needs to be first created during a call to create_pool, i.e. a call to malloc to create the initial list node which will contain the same start address and size of the private heap and that this space is currently free.

create_pool receives an integer representing the requested size for a heap and performs the following tasks:

- checks the heap_list pointer and heap and if still active should do nothing, otherwise do the following:
    - allocate memory for the heap (using malloc)

- checks that the heap space has been allocated (heap pointer is not NULL) and if so, creates the first node of the heap_list and sets its fields to the appropriate values.

my_malloc must search the block list for the FIRST free block that is big enough to satisfy the requested size. If the first free block is more than big enough, then my_malloc should use the first part of the block to satisfy the request and keep the latter part in the private heap. (For example, if my_malloc is looking for a block of 200 bytes and the chosen free block has 1000 bytes starting at address 6000, then the allocated block returned to the caller must be 200 bytes starting at address 6000. A second block of 800 bytes starting at address 6200 is kept in the private heap for possible later use.)

my_free should return a block to the private heap, making that storage once again available. In the process you must recombine the block with any neighboring blocks that are also free.

free_pool() completely frees the private heap. Be sure all allocated memory is freed when this function is called by the client. This is important for multiple tests run while grading. Many students make errors where subsequent tests are rendered useless. Adding the free_pool function to your private heap package affords the ability to start over whenever an error arises during testing. The tests used will call this function whenever an error occurs followed by a call to create_pool to start over with a clean heap for continued testing.

Calls to my_malloc that cannot be satisfied should return NULL. Calls to my_free that provide an invalid pointer should do nothing. **Put your implementation in a file called mallok.c**

In a file called mallok_test.c, write code that will test your implementation. Your testing must include at least the following scenarios (call free_pool after each of these preliminary simple tests):

- create a pool of 1000 bytes. Count how times you can successfully request a block of size 10.

- create a pool of 1000 bytes. Make 5 requests for blocks of 200 bytes. Free all 5 blocks. Repeat this request/free pattern several times to make sure you can reuse blocks after they are returned.

- create a pool of 1000 bytes. Make 5 requests for blocks of 200 bytes. Free the middle block. Request a block of 210 bytes (it should fail) Request a block of 150 bytes (it should succeed) Request a block of 60 bytes (it should fail) Request a block of 50 bytes (it should succeed) etc.

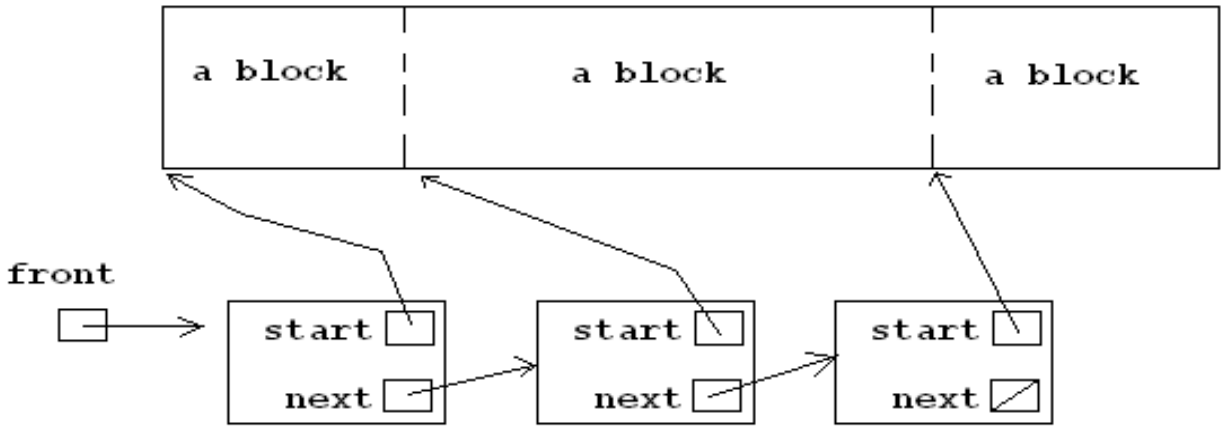
- create a pool of 1000 bytes. Request 5 blocks of 200 bytes. Fill the first block with the letter 'A', the second block with 'B', etc. Verify that the values stored in each block are still there. (Is the first block full of A's, second block full of B's, etc.)

- create a pool of 1000 bytes. Request and return a block of 1000 bytes Request and return four blocks of 250 bytes Request and return ten blocks of 100 bytes

The above tests are only a starting point. The grading tests will make many calls to create_pool, free_pool, my_malloc and my_free with much segmentation and small non-contiguous free blocks generated. For this reason, you should design many more tests than listed above.

Your test code will make many calls to my_malloc, often saving the returned pointers and then using those pointers to call my_free. Your test code should print out messages indicating if the different tests are working as expected.

Here is a diagram showing aspects of the implementation. The private heap is shown at the top. Although dotted lines are shown, it is actually one large piece of memory that was obtained from a single call to malloc during execution of create_pool. The nodes of the linked list have additional fields that are not shown.



There are many factors to consider:

- create_pool should do nothing if the private heap is still active.
- my_malloc should do nothing if the private heap is not active (doesn't exist) or if the requested bytes are not available.
- my_free should do nothing if the received pointer (block) does not exist.
- free_pool should place the private heap in the same state as when the program first began execution, i.e. "ALL" blocks are freed, "ALL" pointers are freed which includes the entire list AND the private heap itself. Any time this function is called, no heap operations can be performed until a new heap is requested through a call to create_pool
- **Be ABSOLUTELY sure to ALWAYS set any pointers to NULL following a system call to free(p), i.e. you will be using malloc for various reasons and free to free up space on the actual heap. Not doing so will cause many of the grading tests to fail and will result in high loss of grade points.** This warning suggest that your various functions should ALWAYS be checking for the validity of a pointer before proceeding with any operations. For example, what would happen if the client calls my_malloc BEFORE ever calling create_pool, wants to start over by calling free_pool followed by create_pool, calls free_pool and immediately calls my_malloc, etc.

Finally, this assignment requires you to use a make file (discussed in class and information is available in the Generic Linked List zip folder on Canvas). This should contain 3 Rules:

- Creation of the final executable file (use myheap as the target which means the executable will be named myheap)-depends on mallok.o mallok_test.o
- Creation of mallok.o-depends on mallok.c and mallok.h
- Creation of mallok_test.o-depends on mallok_test.c mallok.c mallok.h

As with HW6, your program should compile at the command prompt and your code files (mallok.h, mallok.c, and mallok_test.c and a make file for compiling and linking - 4 files in all) should be zipped into a single zipped folder using the Windows zip utility (select files, right click one of the selected group, point to Send to… and click Compressed (zipped) Folder. **Any files included other than listed here will incur a 15 point penalty.** Upload this zipped folder to the link for Home work 7. Also, follow proper documentation rules. **Be sure to name all files and functions EXACTLY as named here.**