CSCI 2271 Computer Systems
Assignment 7: Garbage Collection
Due Sunday, April 7

In this assignment, you will modify your heap manager from HW 6 to support garbage collection. In particular, you should write a file named *gc.c*, which contains the function *gc*.

You should download the file *gc.h*, which contains the declaration for the function *gc*. Basically, this function takes no arguments and returns no value. Its job is to determine which blocks in your heap are inaccessible from the stack, and then to deallocate those blocks. The function should also coalesce unallocated blocks after it finishes.

You should implement the *gc* function using the mark-and-sweep algorithm discussed in class. There are three issues that need more explanation:

- how to find the root nodes;
- how to determine if a root node is a pointer to the heap, and if so, how to determine which block it points to;
- how to mark a block;

Root Nodes

A root node is a value on the stack that points into the heap. Think about the stack at the time *gc* is called—it will contain several stack frames, with the stack frame for *main* at the bottom and the frame for *gc* at the top. A root node can appear in any of these frames. Thus *gc* has to search all of the stack frames, looking for values that point to the heap.

Of course, *gc* cannot possibly know which stack values are pointers. The solution is to assume that every value on the stack is a pointer. That is, your code should treat the stack as an array of 4-byte pointers. Thus you can start at the lowest address, and increment it by 4 bytes each time, until you get to the highest address. Treat each of these addresses as a potential root. This strategy is called *conservative garbage collection*, because it might find a value that looks like a pointer but really isn't. In this case, you might mark a block incorrectly. But that just means that the block will not be garbage collected when it ought to, which isn't a big deal. On the other hand, we know that if a block *is* garbage collected, then it really is garbage.

It is easy to determine the topmost variable on the stack—all you have to do is create a local pointer variable in *gc* and get its address. It is more difficult to determine when you have reached the bottom of the stack. You need a function, called *stackBottom*, to do this for you. When called, this function returns a pointer to the bottom of the stack. This function is in the file *stackBottom.c*, which you can download from Canvas. Please cut and paste this code into your *gc.c* file.

Pointers to the Heap

For each value you grab from the stack, you need to determine if it points to the heap. Use the functions *firstBlock* and *lastBlock* that you wrote for HW 6—If the contents of the pointer is between these two addresses, then it points to the heap.

If you discover a pointer to the heap, you need to traverse the heap's implicit block list to determine which block it is in. That block needs to be marked. You will also need to scan the values in this block to determine if it contains pointers to other blocks.

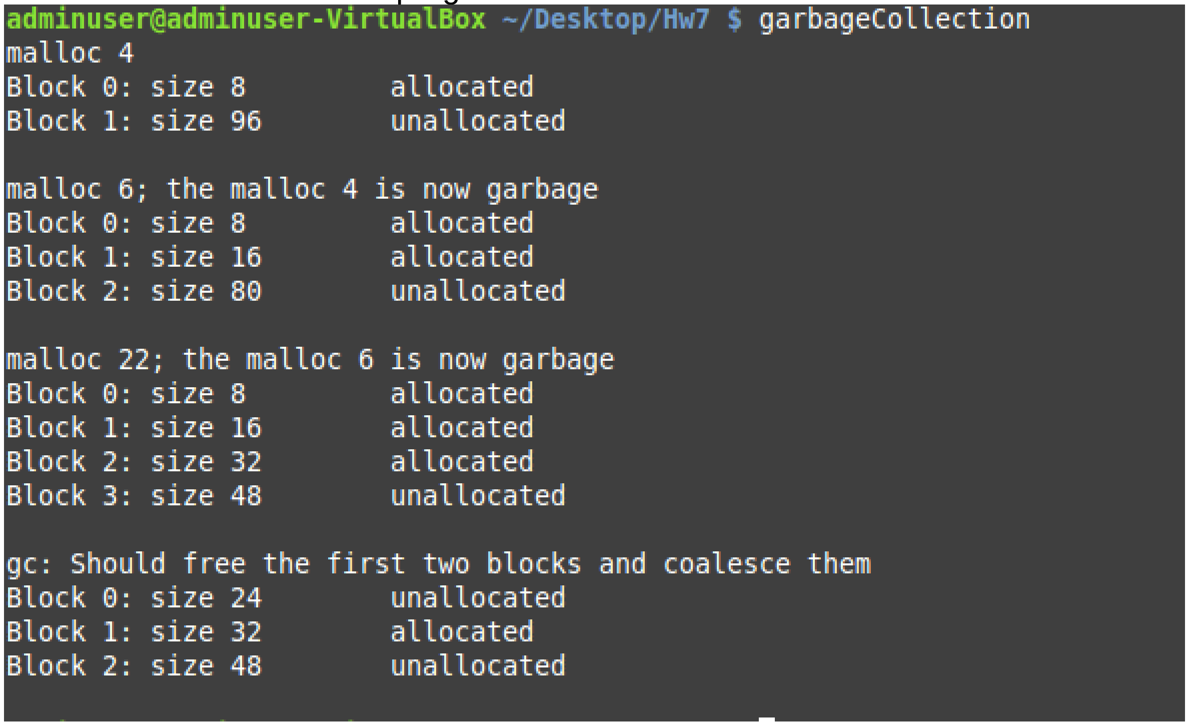
Marking a Block

Recall that the size of a block is divisible by 8, and so the rightmost 3 bits will always be 0's. We are already using the rightmost bit to encode allocation. Use the 2nd bit from the right to encode marking. That is, a block is marked if the value of the block header (mod 4) is either 2 or 3.

Actually, it can be simplified more. Since a block can be marked only when it is allocated, we can assume that a block is marked if the value of its header (mod 4) is 3.

To run the program you need to compile three C files together: *gc.c*, *heapmgr.c* (from HW 6), and *hw7test.c* program. You can download the client *hw7test.c* from Canvas.

Here is a screenshot of the program in action.



```
adminuser@adminuser-VirtualBox ~/Desktop/Hw7 $ garbageCollection
malloc 4
Block 0: size 8         allocated
Block 1: size 96        unallocated

malloc 6; the malloc 4 is now garbage
Block 0: size 8         allocated
Block 1: size 16        allocated
Block 2: size 80        unallocated

malloc 22; the malloc 6 is now garbage
Block 0: size 8         allocated
Block 1: size 16        allocated
Block 2: size 32        allocated
Block 3: size 48        unallocated

gc: Should free the first two blocks and coalesce them
Block 0: size 24        unallocated
Block 1: size 32        allocated
Block 2: size 48        unallocated
```

When you are finished, zip your files *heapmgr.c* and *gc.c* and submit the zip file to Canvas. If you do not have a working version of *heapmgr.c* from HW 6, email me and I can send you the file.