HW4: A Garbage Collector

Homework # 4 A Garbage Collector

**Background**

Dynamic memory (i.e. memory accessed via the malloc and free family of commands) is essential to many programs in the freedom, control, and interpretability it allows. Section 9.9 discusses 2 different types of management for dynamic memory: implicit allocators and explicit allocators.

This assignment focuses on the implicit allocator, perhaps better known as the *garbage collector,* where we don't need to explicitly free the unused memory. Intuitively, we want this tool to find the blocks (or chunks as we refer to them) that are not being used and return them to the process as free chunks.

For further reference, please see section 9.9-9.10 (and specifically 9.10.2) of the textbook.

**Memory layout review**

As a quick review, we can visualize dynamic memory (aka the heap) as a list of chunks. From class, we know that each allocated or free block is started with header (shown as 'h' in the figure below), contains a payload ('x'), and ends with a footer ('f'). (Figure 9.39: Format of heap block that uses a boundary tag.)

----------------------------------------------------------------------

|h|x|x|f|h| | | | |f|h|x|x|x|x|f|h| | | | |...

----------------------------------------------------------------------

^ ^ ^ ^

| | | |

a b c the rest of the heap...

(inuse) (free) (inuse)

In the figure, 'a' represents an allocated section of the heap with a payload + padding taking up 2 words worth of space. Similarly, 'b' is a free node of size 4 words.

**Finding unreachable blocks**

To perform garbage collection, the garbage collector needs to know what blocks are still being used (live) and what are unreachable (dead). To be specific, the dead blocks are considered in-use by the memory allocator but they have been abandoned by the application.

Suppose we are managing a naive hash table (and nothing else) where each bucket contains a chain of items, each created by a malloc(). The dead blocks can be identified by obtaining the list of all in-use blocks and removing the objects on the hash table.

In the example, the buckets of the hash tables contains chains of live blocks. To perform the mark-and-sweep garbage collection, we need to:

(1) MARK: traverse the hash table and mark all the blocks.

(2) SWEEP: scan the list of blocks managed by the heap memory allocator. Every memory block that is not marked shall be freed.

In a more general case, the application maintains a set of root pointers to the directed graphs of live node.

**Your garbage collector**

In this homework, we will build a simple garbage collector for C programs. Starting from the set of root pointers, we traverse the object graphs (actually singly-linked lists) to find all chunks reachable from the roots. These are marked using the second lowest order bit in the header of each chunk. We have implemented several helpful functions for you ranging from checking for marked chunks to navigation of the heap. Please read the skeleton code to get a feel for what is available and what you should be doing.

As of the release of this assignment, we have not covered the concepts related to garbage collection in class; to start on this homework, take a look at section 9.10, and 9.10.2 specifically. The labs on 7/15 and 7/20 will help you get started on the assignment as well.

**The skeleton code**

**The memory allocator**

The textbook provides us a small but fully functional memory allocator implementation:

* memlib.c explicitly manages the heap space. Its job is just to request more pages from the OS and extend the size of the heap memory.
* mm.c contains the actual allocator code that manages the blocks within the available heap space. When it needs more space, it asks memlib for it.

**The application**

The main function in main.c performs a lot of memory allocations but rarely free them. It tries to remember some of the allocated blocks in the root table, but can still get things lost for various reasons. Imagine there is a long-running process that consistently makes these mistakes, the process will quickly build up garbage and eventually get itself terminated by the OOM (out-of-memory) killer.

**The garbage collector**

To save it from creating long-lasting leaks, you will need to implement the garbage collector in hw4.c, which will interact with the memory allocator and the root table to find and free the dead blocks. In the skeleton code, the function gc() does nothing.

To compile the template, type make, and run it with ./hw4

To get started, read main.c and see what's wrong in the output.

Expect to spend a lot of time to read and understand the code. To successfully implement the garbage collector, you will have to understand everything in main.c, hw4.c, and memlib.c, and at least 50% of mm.c.

**Requirements**

Your code goes to the end of hw4.c, starting from Line 75. The autograder will build hw4 using your hw4.c and all of the other original skeleton code. Any changes to the other files will be ignored.

Runtime performance is not a major goal for this homework. However if your program is taking more than 2 seconds, you almost certainly did something wrong.

Finally, make sure you do not output anything extra in your implementations. This will break the autograder.