Programming Assignment #5

CS 6353 Section 1, Spring 2016

In this programming assignment, you will build a basic implementation of a copying garbage collector for C programs, in C (NOT C++).

As usual, I will grade your program by running it against a large test suite. You will be graded by the percentage of test cases you get correct.

The Due Date for this assignment is May 12th at 8:00am.

1. Program Specification

In this program, you will not be writing a stand-alone program. You will be writing functions that I will call from my own C code. The functions you will implement are:

```
void    add_root  (void *addr);
void    del_root  (void *addr);
void   *allocate  (size_t request, size_t *collected);
size_t  heap_max  ();
```

They are described below.

Your implementation will require you to create TWO "heaps" internally. You will only allocate out of a single heap at a time. For the sake of this assignment, you will create two heaps of fixed size, each of which should be 2^{26} bytes (a mere 64MB – good enough for our purposes). This isn't hard, you may do it in one of two ways:

1. Create a global variable of the desired size (less desirable, but easy).
2. Use the `mmap()` system call to get virtual memory from the OS (use the `MAP_ANONYMOUS` flag). This is a little harder, and since you're not given an initialization routine, you would have to add logic to do this in `allocate()` if there is no heap.

You will NOT keep a "free list" like you did with Assignment 3. ARE expected to be able to re-use memory as I use your library to allocate and deallocate items. In fact, you must NOT provide a `deallocate()` function (per the last assignment)—because your program is going to automatically garbage collect!

As with the last example, you are not to use any heap-allocated memory except that you may use `mmap()` to establish your own heaps.

void add_root(void *addr, size_t len);

This function registers a root (starting point) for your garbage collection algorithm. The first argument is a pointer to memory, and the second is the number of bytes in the root. I will call it in the following circumstances:

1. At the beginning of my program, once for each global variable that can possibly hold a pointer. Note that, if I have a global variable that is an array of data, there may be pointers into the heap anywhere in that array that is word-aligned to a 64-bit word.
2. When I create a new activation record / stack frame (i.e., when I call a function), I will call it for any local variables or parameters that may possibly hold a parameter.

void del_root(void *addr);

This function removes a root—you should NOT use it in garbage collection. I will call this once for each root when the root goes out of scope (when I return from a function).

void *allocate(size_t request, size_t *collected);

The allocate function must return a pointer to memory from within your heap. The amount of memory available to the user must be a minimum of `request` bytes. The address you return to the user must be word-aligned to 64-bit words—meaning that the memory address must be divisible by 8 without a remainder.

If you cannot return enough memory for the programmer to use out of the current heap, then you must garbage collect, as described below. If garbage collection does not free up enough memory to service the request, then return the `NULL` pointer, and write `0` into the `collected` field.

Otherwise, if you did NOT garbage collect, write 0 into the `collected` field. If you did garbage collect, then write the number of live objects you copied into the TO-space. A live object is the result from a previous call to `allocate()`, which is still accessible from the roots, and thus lives in the TO-space.

Your `allocate()` function MUST ONLY garbage collect if there otherwise wouldn't be enough memory to service the request.

For garbage collection, you MUST implement the compacting, copying collection algorithm that we discussed in class. When it is time to do garbage collection, you must copy all live data from one of your heaps into the other, re-writing any pointers to point properly into the new heap.

You can make the following assumptions:

1. The program will not under any circumstances be multi-threaded.
2. The user will only store pointers that are word-aligned, and only in roots or memory created via `allocate()`.
3. Pointers might point into non-byte aligned memory.
4. If a pointer points anywhere into the result of a previous call to `allocate()`, then that entire `allocate()` result should be preserved in the new heap.

There are a few subtleties that are worth mentioning:

1. You may have multiple roots, which we did not explicitly talk about in class. You can handle this by copying all the allocated chunks from the FROM-space into the TO-space that are (in the FROM-space) directly addressable from the roots before you follow any pointers.

2. Since a pointer alone doesn't tell you how much to copy into the TO-space, you will still need to keep a header for each allocation (as in the previous assignment, it should be stored BEFORE the pointer you pass back to the caller). The header is a good place to put the "forwarding pointer".
3. Since a pointer might point into the middle of an allocation, you may need to scan backwards or otherwise search to find the start of the allocation to copy out. You also need to be careful about how you re-write this pointer (i.e., remember that it's offset from the start of the allocation).
4. A pointer that points into your heap may be pointing to unallocated data (for instance, if there was a buffer overflow, or a math error). In such a case, you must NOT copy any allocation. Instead, leave the pointer untouched, as if it were data.
5. A single allocation or root may contain multiple pointers into the heap.

size_t heap_max();

This call should return the largest number of bytes I can request from your `allocate()` without triggering garbage collection.

2. Other Requirements

You will receive a 0 on this if any of these requirements are not met!

1. The homework must be written entirely in C, with no third party libraries.
2. You must submit either a ZIP file containing only: .c files, .h files and a README, OR a single .c file.
3. Your homework must NOT have a `main()` function. You're basically just delivering me a library.
4. I will compile your code using the command: `cc -c *.c`
5. I will then link test cases against your code, and run them using your memory allocator.
6. Your code must compile, run and work properly in the reference environment. Even if it works on your desktop, if it doesn't work in the reference environment, you will get a 0.
7. You must submit your homework before 8:00am on the due date.
8. You must submit the homework through the course website, unless otherwise pre-approved by the professor.
9. You may not give or receive any help from other people on this assignment.
10. You may NOT use code from any other program, no matter who authored it. This includes 3rd party libraries.

4. Example

Here's a simple example of how I might use your code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int     x;   // DO NOT REGISTER, it cannot contain pointers.
char *strings[4] = {"A", "B", "C", "D"};
typedef struct tree_node_st {
  struct tree_node_st *sibling;
  struct tree_node_st *child;
         char *value;
} node_t;

node_t *tree_top;

static void register_roots() {
  add_root(&strings, sizeof(strings));
  add_root(&tree_top, sizeof(tree_top));
}
```

```
static void init_tree() {
  tree_top = alloc_node();
  tree_top->value = "root";
}

static node_t *alloc_node() {
  node_t *result;
  size_t  status;

  add_root(&result, sizeof(result));
  result = (node_t *)allocate(sizeof(node_t), &status);
  if (!result) {
    printf ("Out of memory.\n");
    exit   (0);
  }
  if (status) {
    printf ("Nice!  GC ran and collected %zu bytes for us.\n", status);
  }
  result->sibling = 0;
  result->child   = 0;
  result->value   = 0;
  del_root(&result);
  return result;
}

static char *alloc_copy_str(char *s) {
  char *result;
  size_t status, i, l = strlen(s) + 1; // Copy the null.

  add_root(&result, sizeof(result));
  result = (char *)allocate(sizeof(node_t), &status);
  if (!result) {
    printf ("Out of memory.\n");
    exit   (0);
  }
  if (status) {
    printf ("Nice!  GC ran and collected %zu bytes for us.\n", status);
  }
  for (i=0;i<l;i++) result[i] = s[i];
  del_root(&result);
  return result;
}

static void build_tree(int argc, char **argv) {
  node_t *cur = tree_top;
  node_t *next;
  int     i;

  add_root(&cur, sizeof(cur));
  add_root(&next, sizeof(next));
  for (i=1;i<argc;i++) {
    if (!strlen(argv[i])) continue;
    next        = alloc_node();
    next->value = alloc_copy_str(argv[i]);
    if (argv[i][0] >= 'A' && argv[i][0] <= 'Z') {
      cur->sibling = next;
    } else {
      cur->child   = next;
    }
    cur = next;
  }
  del_root(&next);
  del_root(&cur);
}

int main(int argc, char **argv) {
  register_roots();
  init_tree();
  build_tree(argc, argv);
}
```

5. Reference Environment

The reference environment is once again kimota.net. Your same login credentials will continue to work.